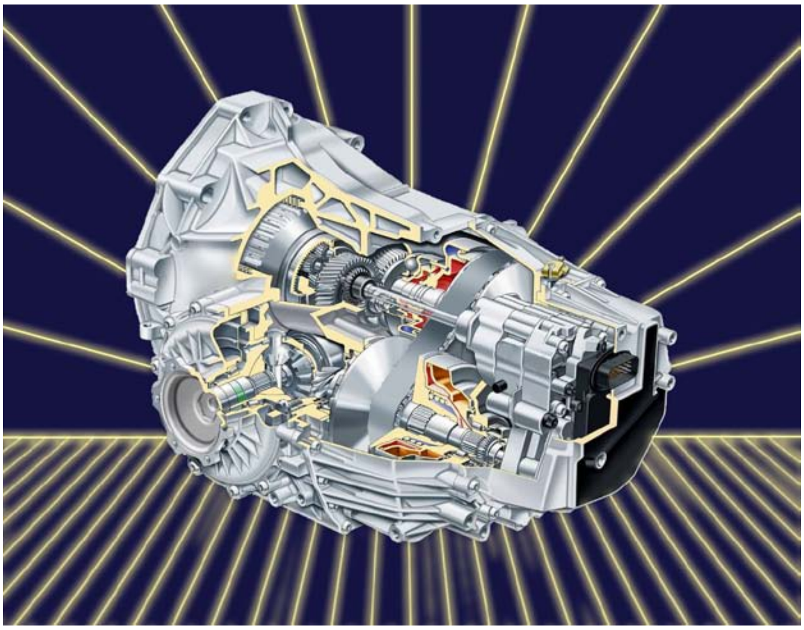


基于 Linux 2.6.18 内核源码

V0.1

Linux内核协议栈源码解析



luoyu

目　　录

图目录

表目录

前　言

这是一本介绍协议栈源代码的书，不是介绍“协议”的书。对协议非常感兴趣的读者，本书是不太适合的，要看最好是去看 RFC。由于只分析内核最主要的代码，所以提前跟读者打招呼：本书不是圣经。

没有写书的经验，没有流畅的表达，只有比较随性的记录，这就是本文档——或者本书的整体风格。我的目标读者是广大对 Linux 和 IP 协议栈实现感兴趣的读者。为什么要写这么一本书，抛砖引玉也。大家都知道 Linux 内核已经发展到了 2.6.26，其在服务器已经证明其设计的精巧和健壮，当内核也从非抢占式发展成为抢占式后，嵌入设备市场上也将要掀起一股风浪。于是 Linux 内核分析的资料层出不穷，但有的太老（内核的代码还使用 2.2 的），而有的对网络部分不甚详细，于是笔者萌发了分析整个 Linux 网络协议栈的想法。希望能在研究一些经典代码时发现与时俱进的部分，抛砖引玉，更多的人参与到研究网络协议栈的实现技巧上——而不是协议的 RFC 文档。

由于本人在网络方向做过一些开发，所以对网络内部实现很感兴趣，于是在利用学习和工作时间之余记录了分析和调试 Linux 及 FreeBSD 的网络协议栈。但本人较懒，本来应该于 2006 年就可以写出这本书，但由于中国学生的特殊情况，我浪费了很多时间，后来再由于中国软件工程师的特殊情况，我又虚度了很多光阴，直到看见《深入理解 Linux 网络内幕》面市，本人才大呼晚亦，惭愧啊。但我想许多有心写书但无力/时/财的同行们应该和我有相似的感叹，我们咋就没有这么多时间和注意力放在我们热爱的事业上呢？成天为了一些琐碎却又不得不做的事情头疼不已，突然有一天发现自己变老而一事无成时，心中苦闷谁人解？

本人从 2005 年计划写这本书，当时研究的内核版本是 2.6.5，但是 3 年后我使用了 RHEL5 作为研究平台，内核却升级到了 2.6.18，就协议栈的部分代码就已经发生了很多变化，尽管最新的内核版本是 2.6.26 版本，但我还是锁定了这个版本，毕竟是服务器使用的版本呀。本书的编写计划也和 Linux 内核的发行版本一致，即先出 0.1 版，包括了一些基本框架就发布出来，收集读者的意见，修改一些错误的地方，然后出 0.2 版，在此基础上再增加一些部分，比如 SCTP、无线等内容，形成 0.3 版，以次类推，达到大部分人可以接受的程度。

由于本人还有一份不体面也没有前途的职业，工作会越来越忙，只能在业余时间赶着写这本书，所以造成本文档有不少遗漏甚至误解，希望大家在拍砖的时候轻点砸。最后，我的语言功底一般，如果同行们见到错误不要太苛责于我，毕竟这是我的第一篇文章。能提出宝贵的意见，我绝对能洗耳恭听。

我的邮箱是 blade_ly@yahoo.com.cn
欢迎来信探讨计算机相关技术，不限 Linux，不限协议栈。

感　　谢

写一本书真累，真的，我很佩服安德鲁·坦尼伯姆(Andrew Tanenbaum)、W.Richard Stevens、侯捷等等技术牛人能写出那么多经典书，他们是我们一生追求的标杆，让我心里还有一丝梦想。所以感谢它们。

感谢应该感谢的身边人，我不喜欢肉麻，具体不说是谁谁谁啦。

第1章　协议栈概述

1.1　操作系统及网络协议介绍

1.1.1　Linux 操作系统架构简介

可以说，Linux 是 21 世纪初最火的操作系统。注意，我只在这时说它是最“火”的，而不是最“好”的。最好的定义对于每个人都不一样，为避免产生口水仗，我不在书中对 Linux 进行评价。不过我得先介绍一下 Linux 的架构。

Linux 肯定是一款大内核操作系统，Linus Tovals 和 Tanenbaum 的网上争论余音绕梁，相信知道此事的读者一定还记得 Linus 支持大内核的建议吧。虽然说 Linux 是以大内核的方式运行，但编译方式已经吸收了 Windows 的动态加载模块的特点。也就是说，彼时（80‘s）的大内核和现在说的大内核不完全是一个概念。那时候的大内核，不仅运行的时候所有驱动、文件系统、包括本书要讨论的协议栈要运行在内核态，而且编译的时候，编译的文件必须同时被编译到一个大的二进制文件中。如今的内核，在编译的时候，可以有选择的加入或减去某部分代码，使其编译出来的内核变“小”，而一些驱动程序和模块可以在系统起来以后再加载。那么就单纯的那个内核镜像而言，真的还不算“大”，这个时候要说 Linux 是大还是小还真有点难。

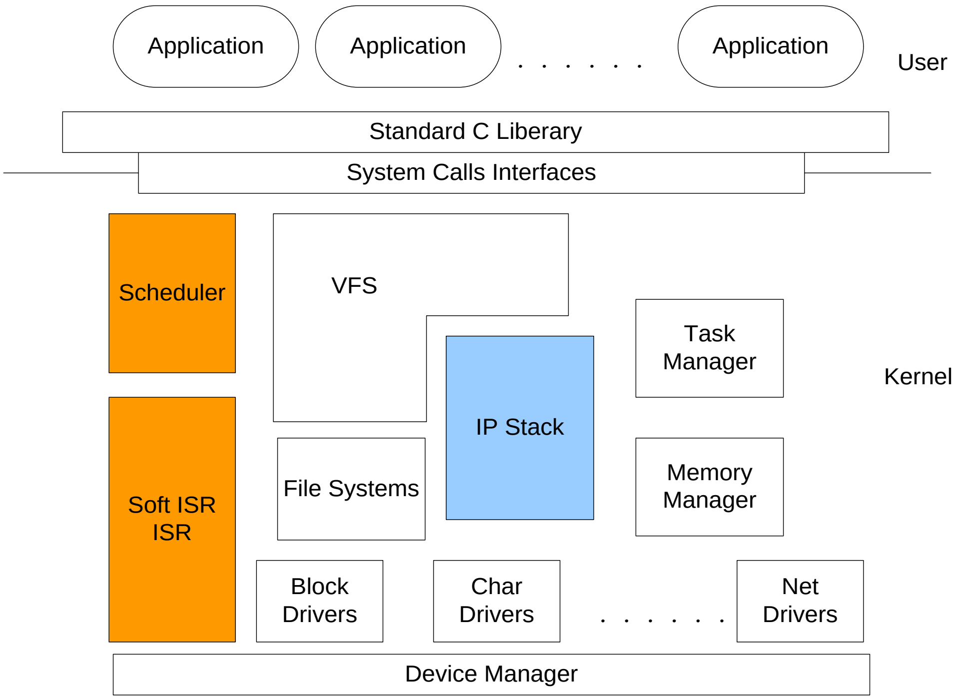
> 操作系统内核可能是微内核，也可能是大内核（后者有时称之为宏内核 Macrokernel）。按照类似封装的形式，这些术语定义如下：
>
> - 微内核（Microkernel kernel）——在微内核中，大部分内核都作为独立的进程在特权状态下运行，它们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，如果在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其它进程（或模块）通讯以完成所需任务。 在这些设计中，微内核部分经常只不过是一个消息转发站：当系统调用模块要给文件系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离。（某些时候，模块也可以直接给其它模块传递消息。）在一些微内核的设计中，更多的功能，如 I/O 等，也都被封装在内核中了。但是最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就可以完成将整个内核移植到新的平台上。其它模块都只依赖于微内核或其它模块，并不直接直接依赖硬件。 微内核设计的一个优点是在不影响系统其它部分的情况下，用更高效的实现代替现有文件系统模块的工作将会更加容易。我们甚至可以在系统运行时将开发出的新系统模块或者需要替换现有模块的模块直接而且迅速的加入系统。另外一个优点是不需要的模块将不会被加载到内存中，因此微内核就可以更有效的利用内存。
> - 大内核（Monolithic kernel）——单内核是一个很大的进程。它的内部又可以被分为若干模块（或者是层次或其它）。但是在运行的时候，它是一个独立的二进制大映象。其模块间的通讯是通过直接调用其它模块中的函数实现的，而不是消息传递。
>
> 单内核的支持者声称微内核的消息传递开销引起了效率的损失。微内核的支持者则认为因此而增加的内核设计的灵活性和可维护性可以弥补任何损失。就像 Linux 内核是微内核和单一内核的混合产物一样。Linux 内核基本上是单一的，但是它并不是一个纯粹的集成内核。为什么 Linux 必然是单内核的呢?一个方面是历史的原因：在 Linus 的观点看来，通过把内核以单一的方式进行组织并在最初始的空间中运行是相当容易的事情。这种决策避免了有关消息传递体系结构，计算模块装载方式等方面的相关工作。（内核模块系统在随后的几年中又进行了不断地改进。）
>
> 如果 Linux 是纯微内核设计，那么向其它体系结构上的移植将会比较容易。实际的情况是，Linux 内核的移植虽然不是很简单，但也绝不是不可能的。虽然这比微内核的移植需要更多的代码，但是 Linux 的支持者将会提出，这样的 Linux 内核移植版本比微内核更能够有效的利用底层硬件，因而移植过程中的额外工作是能够从系统性能的提高上得到补偿的。
>
> 这种特殊设计的权衡也不是很轻松就可以达到的，单内核的实现策略公然违背了传统的看法，后者认为微内核是未来发展的趋势。但是由于单一模式（大部分情况下）在 Linux 中运行状态良好，而且内

核移植相对来说比较困难，但没有明显地阻碍程序员团体的工作，他们已经热情高涨地把内核成功的移植到了现存的大部分实际系统中，更不用说类似掌上型电脑的一些看起来很不实际的目标了。只要 Linux 的众多特点仍然值得移植，新的移植版本就会不断涌现。

这不是自相矛盾吗？有的读者可能会疑惑了。其实，大和小内核之争已经过时了，连 Windows 也号称过微内核，那么有必要这么认真吗？其实只要记住，除了嵌入式，没有一款商用（包括开源）的操作系统是以真正学术上的微内核方式存在。也就是说，所有的商用操作系统都是把驱动、文件系统、协议栈等塞入内核之中，原因在于安全和效率。这就不必多说了。

把这些模块放入内核中确实提高了安全性和效率，但也引入了新的问题，比如说进程之间的调度，进程之间同步互斥等。Linux 内核在发展初期由于没有考虑到这些问题，以至于在实时性能上为人诟病。不过本书的内容与实时性关系不大，如果读者希望能进一步了解的话，可以和笔者讨论。

下图是 Linux 整体的一个层次图：



图表 1-1 操作系统架构图

从上图可以看出，IP Stack 就是本书要分析的模块，坦白地说，这个模块并不是一台机器所最需要的部分，因为并不是每个机器都要上网、聊天、收发邮件、联机对战。。。。。什么？还有不能聊天的机器吗？嗯，笔者的机器目前正是如此。

从这个模块的位置来看，它牵扯到内核中大部分模块，可以说不了解其他任何一个部分，对了解协议栈的工作行为都有困难。这也是 IP Stack 的难点：拿出 rfc 去单独理解 IP 协议，没有太大困难，但要真正看懂 Linux 是如何实现的，则需要比较广博的操作系统知识。

了解了 IP Stack 在操作系统内的位置后，那么我就开始说网络了。

1.1.2 网络协议发展介绍

"IP 化"是这两年比较热门的一个概念。从计算机发展到如今，实际上曾经出现过多种网络协议，从物理层到应用层，无数公司发明了自己的网络协议，其实有的性能还不错，只是由于生不逢时或商业原因，消失在大家的视野。而 IP 协议，由于其健壮性和简单性，被军方和学校发展成为一套协议族，就是我们常说的 TCP/IP 协议族。TCP/IP 本身是一个协议族，还包含了 ARP，ICMP，UDP 等协议。它从开始提出到现在的广泛使用，已经差不多 30 年了，在这段时间内，不管是大型组织还是公司都提出其他类型的网络协议栈试图取代 TCP/IP，但都没有成功，反而是 IP 协议逐渐蚕食其他网络市场的分额，比如 IPX。

那么 IP 网络的优点在那呢？正如我前面所说，它简单。是的，它对应用人员来说比较简单，但代码可不简约。它是一堆小协议和数据的集合，实现的复杂给人带来的好处就在于管理简单，因为"智能"的部分它来做了。IP 协议栈的中心就在于，使用 IP 协议将数据包发送到任何网络，而且数据包到达的时

机可以不同。

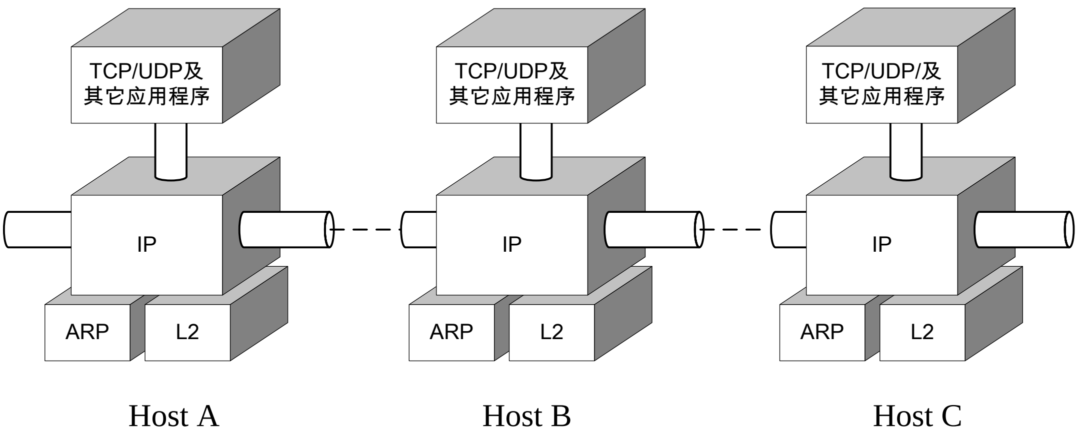
前几年的网络教材上会提到 ISO 提出网络分为 7 层的概念，如下图：

OSI 七层模型图解



因为事实上的网络协议栈实现基本采用 TCP/IP 的 4 层架构，从底往上分别是：链路层，网络层，传输层，应用层。但是在实际开发人员的眼中，却是五层，即在链路层之下还有一个物理层。所以，本书中将链路层简称为 2 层或 L2，网络层简称为 3 层或 L3，而传输层即为 4 层或 L4。

从下图看到，我们的 TCP/IP 协议栈包含了图中若干部分，IP 是连接 2 层和 4 层的中枢，它不仅作为数据在系统（上下方向）中传送的纽带，也作为机器间数据传输（水平方向）的判断者，这就无形中增加了实现 IP 层的难度。蓝线即表示本机系统的数据流，红线表示机器间的数据流。ETH 代表以太网功能，它就是 2 层功能的一个具体实现。ARP 属于 2 层和 3 层之间的一个连接层，我们可以认为它是 2.5 层的协议。L2 协议包括了 PPP 和 SLIP 及 Ethernet，前两者不属于本书范畴，只需知道它们和 Ethernet 一样，属于底层的传送协议即可。



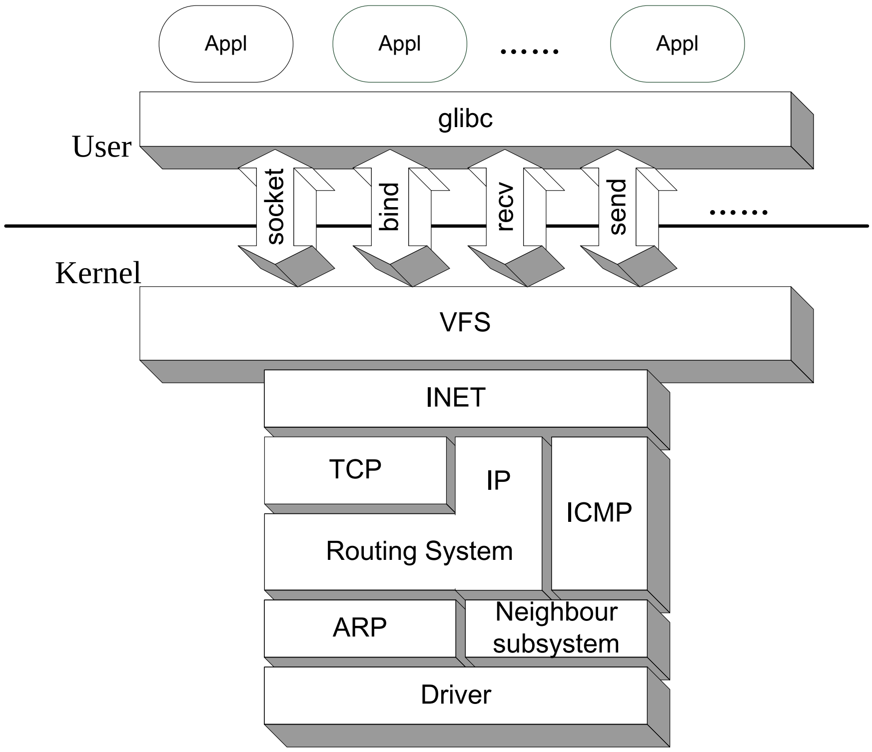
图表 1-2 IP 为什么重要

图中所示 IP 和 IP 相连，实际只是逻辑的概念，实际数据的传输还是通过下层协议再到物理层进行。远非图中画的这么简单。

从上面的图，让我至少想起 2 部关于机器人的电影，一部是 90 年代的《波波 5 号 （其中的主人公就是

这个编号)，第一次看此片时，当我以为波波 5 号被击毁的时候，居然会哭......不可否认的是，它已经影响了我对技术的认知，如果某一天机器人有了人的思维和智慧，怎么办？推广一下，如果上图中的协议栈具备人的感情，那......
另一部是 2008 年上映的《机器人瓦力》，样子也和波波五号类似，也给了我一些感动，推荐！
世界上的每一台机器通过协议栈相连，其基本形式大概也就是上图所示，只不过有的实现是 Linux，有的是 UNIX/BSD，有的是 Windows 系统提供的协议栈，当然还有一般人看不见的网络设备中厂商自己实现的协议栈。

每一类软件模块都不能独立存在，必定依托系统其它模块的支持才能工作，协议栈更是如此，由于协议栈是在内核中实现，所以，必须搞清楚操作系统中是如何支持协议栈的，下图可以作为一个参考：



图表 1-3 真实操作系统协议栈实现

上图对于大多数人比较眼熟，但我还是要着重提到两个不太为人注意的模块，一个是 glibc 库，对于大多数从事软件编程 2 年以下的读者，这个部分比较难理解。其实我们几乎无时无刻不与这个部分打交道，比如说不管是应用层软件开发，还是嵌入式软件开发，我们用过的 malloc 函数、strcpy 函数都是这个库提供的。不仅如此，网络编程中用到的 API 接口也是这个库提供的。至于怎么提供，下文提及。另一个要强调的模块是 INET，它不属于 TCP/IP 体系必须的一个部分，但 TCP/IP 层的接口要通过 INET 层才能访问操作，这一操作是在网络初始化时就已经注册到 BSD 风格的 socket 层的。所谓 BSD 风格就是我们常说的 socket、bind、connect、listen、send 和 recv 等系统接口的调用风格。不管是类 UNIX 还是 Windows 操作系统都必须实现这些接口，这些接口内部不仅支持你上网（就是 AF_INET），还支持你的应用程序之间的通信（比如 AF_UNIX），或者内核与用户之间通信（AF_NETLINK），甚至一些少见的协议（比如 AF_IPX），但就本书的内容显然属于 AF_INET 层的范围，它封装了 TCP/IP 的接口。

从这幅图中可以看到 IP 层并不是完全在 TCP 之下。换言之，应用程序是可以绕过 TCP 层而直接与 IP 层协作，比如我们常用的 ping 命令。

1.2 本书的组织和安排

读者一般都玩过搭积木这种游戏，也都知道积木其实也就是几种类型的“块”组成，在搭积木的时候我们应该关心要建造的城堡的形状而不是积木“块”的材质和硬度等一些非常细节的东西。分析代码也是这样。Linux 内核大量采用了几种通用的数据结构，正如同积木块，如果每遇到一次这种代码我就分析一次，估计本书篇幅会太大，于是单独拿出来先对其分析，记住其基本操作方式和熟悉，只要形成了

思维定势，那么我们对 Linux 内核的理解难度会降低。

1.2.1 基本的数据结构和计算机术语

本书的读者群应该是掌握了基本的 Linux 编程技术并对网络协议有一定了解的中高级开发人员，在讲述各章节的内容中，许多浅显的技术将会一带而过，不花太多的笔墨。但以下一些编码技术和术语，需要着重强调。

- 链表结构

链表数据结构的定义很简单（节选自[include/linux/list.h]，以下所有代码，除非加以说明，其余均取自该文件）：

```
struct list_head {
 struct list_head *next, *prev;
};
```

list_head 结构包含两个指向 list_head 结构的指针 prev 和 next，由此可见，内核的链表具备双链表功能，实际上，通常它都组织成双循环链表。不过它和传统教科书上介绍的双链表结构模型不同，这里的 list_head 没有数据域。在 Linux 内核链表中，不是在链表结构中包含数据，而是在数据结构中包含链表节点。也就是说，开发者预计某种数据结构将要组成链表时，它的成员中必含一个 struct list_head 成员。

- 由链表节点到数据项变量

我们知道，Linux 链表中仅保存了数据项结构中 list_head 成员变量的地址，那么我们如何通过这个 list_head 成员访问到它的所有者呢？Linux 为此提供了一个 list_entry(ptr, type, member)宏，其中 ptr 是指向该数据中 list_head 成员的指针，type 是数据项的类型，member 则是数据项类型，例如，有一个由 proto{}组成的链表，名字叫 proto_list，而每个 proto{}结构中的 list_head{}成员名字叫 node。我们要访问这个链表中第一个节点，则如此调用：

```
list_entry(proto_list->next, struct proto, node);
```

list_entry 的使用相当简单，相比之下，它的实现则有一些难懂，先告诉大家的是，Linux 内核用的 GCC 编译器使用了一些比较新的关键字，比如 typeof，所以下面的代码是无法在 VC 等编译器上通过：

```
#define list_entry(ptr, type, member) container_of(ptr, type, member)

#define container_of(ptr, type, member) ({   \
       const typeof( ((type *)0)->member ) *__mptr = (ptr); \
       (type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

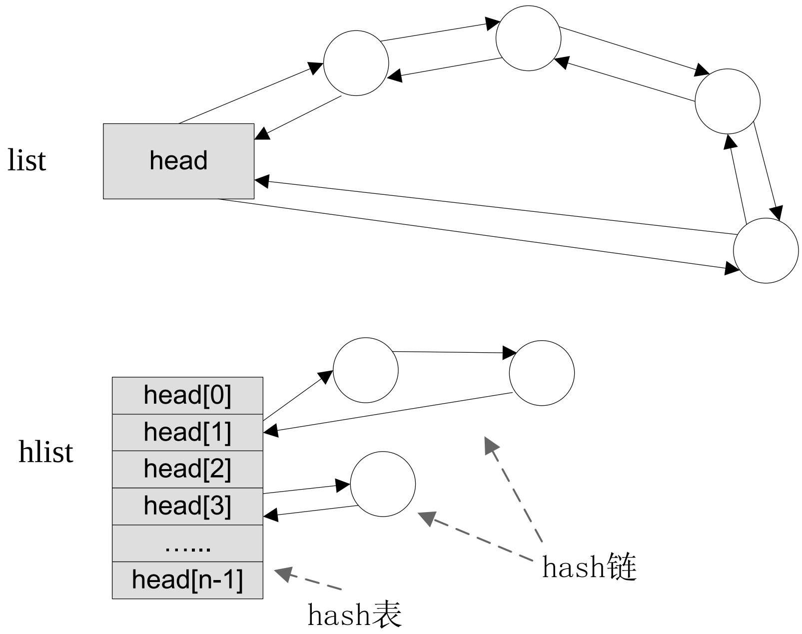
正是利用了 typeof，内核中许多技巧得以实现。，在这里，先求得结构成员在与结构中的偏移量，然后根据成员变量的地址反过来得出属主结构变量的地址。

要记住的是 container_of() 和 offsetof() 并不仅用于链表操作，这里最有趣的地方是((type *)0)→member，它将 0 地址强制"转换"为 type 结构的指针，再访问到 type 结构中的 member 成员。

对于给定一个结构，offsetof(type,member)是一个常量，list_entry()正是利用这个不变的偏移量来求得链表数据项的变量地址。

- hlist

下图是关于 Linux 内核中 list 和 hlist 的区别，hlist 是 hash list 的简称，即用拉链法实现的 hash 数据结构。它由 2 部分组成：hash 数组和冲突链。当节点第一次要插入 hash 表的时候，它必定是先插入 hash 数组中，而以后要插入的节点如果发生了冲突，则可以挂在数组后面，形成一条链表。当然也可以插入数组，原先在数组中的节点被挤出来形成一条链表。两者的区别如下图：



图表 1-4 list 和 hlist 的区别

也许 Linux 链表设计者认为双头（next、prev）的双链表对于 HASH 表来说"过于浪费"，因而另行设计了一套用于 HASH 表应用的 hlist 数据结构——单指针表头双循环链表，

```
struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

代码段 1-1 hlist_head 的定义

struct hlist_head 仅仅用了一个指针，占 4 个字节，因为 hash 数组往往相当大，这样可以节省 4 个字节乘以数据大小的空间。从上图可以看出，hlist 的表头仅有一个指向首节点的指针，而没有指向尾节点的指针，这样在可能是海量的 HASH 表中存储的表头就能减少一半的空间消耗。我习惯上称 hash 数组为 hash 表，而将冲突节点链表称为 hash 链。

不太懂 HASH 表算法的读者还是不要往下看了，Linux 内核特别是网络部分中很多地方是用 hash 表组织的，为了节省您宝贵的时间，您要么把 HASH 算法复习一下，要么把这篇文档删除吧。

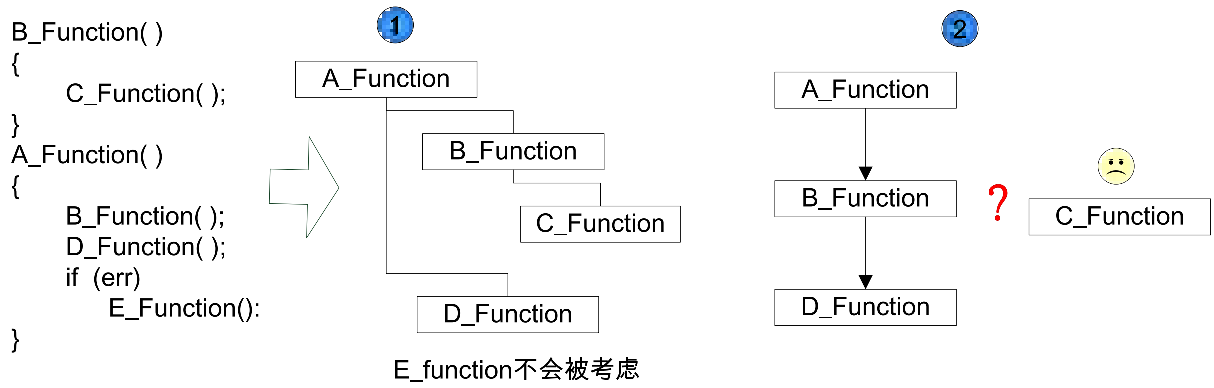
1.2.2 图片风格演示

本书中所有图示都是笔者画出，有一些独特的表示方式需要给读者做实例。在每一章/节的开头，笔者提供了关于此章的函数调用树。这种调用树在代码分析过程中经常可以看到，它简单易懂，从宏观上体现了开发人员的基本思路。对于简单的代码流程可以快速跨过。而对关键的代码段进行分析，指出其算法，给出注释。

这样两种粗细结合的代码分析过程在笔者从事软件开发过程中起了很大的帮助，相信也能给读者以帮助，尽早理解协议栈工作流程。

- 函数调用树

由于本书中不可能把内核中所有代码抄过来，所以对比较简单的代码的分析采用了调用树的方式。比如下面代码：



图表 1-5 函数调用树的演示

对于函数调用，我们采用第①种方法表示。如果采用第②种比较传统的方式，那么C_Function将放在何处呢？而且，在大部分代码中，有很多对错误进行判断的代码分支，如果我们要完全画出这些错误处理分支，那么工作量是非常巨大的，而且是没有必要的，最关键的是影响读者理解主要的脉络。所以本人的开发经验告诉我，传统的表示方法是不适合大范围、大跨步了解一个原有系统结构的。

书中还有一些序列图，我想大家应该不会产生理解上的困难吧。

1.2.3 本书的组织

网络协议栈的内容实际上包括了除 TCP/IP 在内的协议，当前基于 TCP/IP 的协议不下百种，这从一个方面说明了 TCP/IP 网络强大的影响力，也给笔者带来了分析的难度：要不要分析诸如 SLIP, PPP, SIP, OSPF 等协议？要怎样才能将这些协议从 TCP/IP 代码中剥离？如果不介绍这些协议代码会对协议栈的理解有什么影响？

出于能力和时间关系，笔者决定舍弃大部分协议，不在本书中介绍关于驱动程序、包封装的协议、或协议的具体实现。本书侧重介绍的是 Linux 中 TCP/IP 网络协议栈的实现，将讲解各种机制的实现。

本书侧重介绍 Linux 协议栈的 TCP/IP 协议分析。在阅读本书时，希望读者以协议层次之间传递的数据流为根本，作为理解协议栈工作原理的主线，再辅以函数调用序列分析，相信能在阅读本书后对协议栈有清晰的理解。

本书的组织如下：首先介绍操作系统的部分基本知识，然后分析其启动过程，接着重点分析协议栈的初始化，为分析、理解数据流向打下基础。接下来，由简入难，先介绍配置，分析 IP 路由的存储和创建，还会介绍 UDP 和 TCP 内部的实现，最后，引入一些比较流行的话题，比如流控、VLAN、二层协议等。

从一定角度看，Linux 本身就是网络的代名词。它的开发和发展都是通过程序员在网络上，尤其是在互联网和新闻组上交流信息、程序代码完成的。Linux 从 1991 年出现到现在，网络方面的代码一直是至关重要的部分，代码间关系相当复杂。从在内核代码根目录单独建立了 net 目录，足以看出网络在 Linux 中是和文件系统一样重要。当然，网络相关的代码在 drivers/net 中还有一部分。

下面是 Linux 内核源代码的目录树：

```
Linux-2.6.18
    |_arch
    |_configs
    |_documentation
    |_Drivers
    |     |_net
    |     |_......
    |_fs
    |_include
    |_init
    |_ipc
    |_kernel
```

```
|_lib
|_mm
|_net
|      |_802
|      |_802.q
|      |_bridge
|      |_core
|      |_ethernet
|      |_ipv4
|      |_ipv6
|      |_llc
|      |_netlink
|      |_packet
|      |_sched
|      |_......
|_scripts
|_security
|_sound
|_usr
```

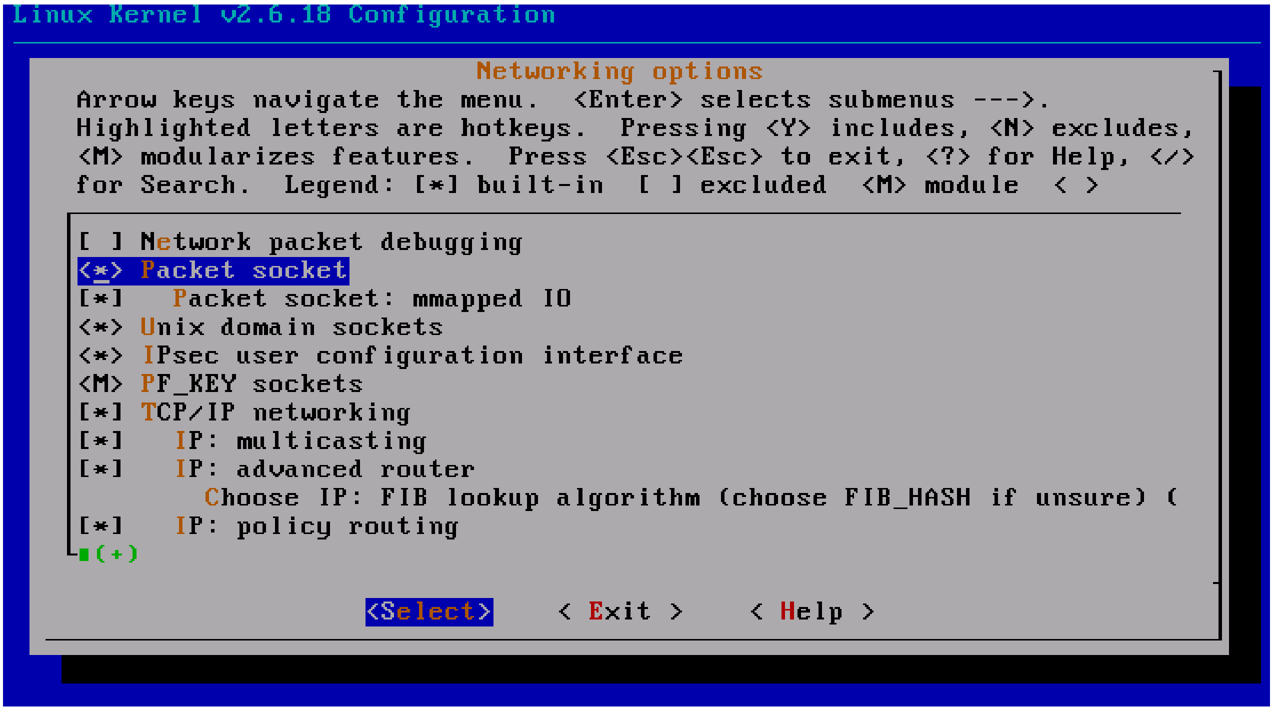
由于网络协议栈涉及的东西实在太多，我只能挑一些最常见的内容去说，而不能面面俱到，因为我还没有那么广的阅历，也没有那么多时间全部写就。我打算这样安排，本书将仿照软件升级出版本的方式，把此书定个版本号。目前就是 0.1 版，当发布到网上时，内容上基本上不会有太多变化，主要是改正错误形成 0.2。而到当发布 0.2 之后，本人将加入 WLAN，SCTP 等较新的内容，当然，我得有时间，呵呵。

第2章　系统初始化

大凡拿到 Linux 发行版的人，必先想编译一次内核源代码以图一快。然后尝试裁减内核，然后查看源码，然后修改，那么我现在先给一个初步的关于 Linux 内核源代码编译，裁减的介绍，因为对于研究内核源代码，不了解其编译的内部机制，就不可能精通其代码执行的起源。

废话少说，来看看如何编译源代码。

首先，进入 Linux 源码目录，键入 make menuconfig，会出现一个类似菜单的界面让用户选择是否裁掉或加进某模块。我们关心的网络模块界面如下图：



图表 2-1Linux 内核编译——网络选项部分

在这幅图中，有的选项前是“*”,有的是“M”，有的是空，这表示什么呢？即当选择“*”的时候，模块被编译进内核，在系统启动的时候，被主调函数调用执行；而如果选择“M”的时候，表示被编译成一个.ko 文件，放在某个目录下，系统启动的时候依靠脚本把这些目标文件装入内核。为什么要有这样的区分？是因为这两种编译方式对决定了内核的大小，也决定了内核模块被初始化的过程，这些内容在其他书中有介绍，下面的章节还会提及。不过，我们还是先来研究系统启动吧！

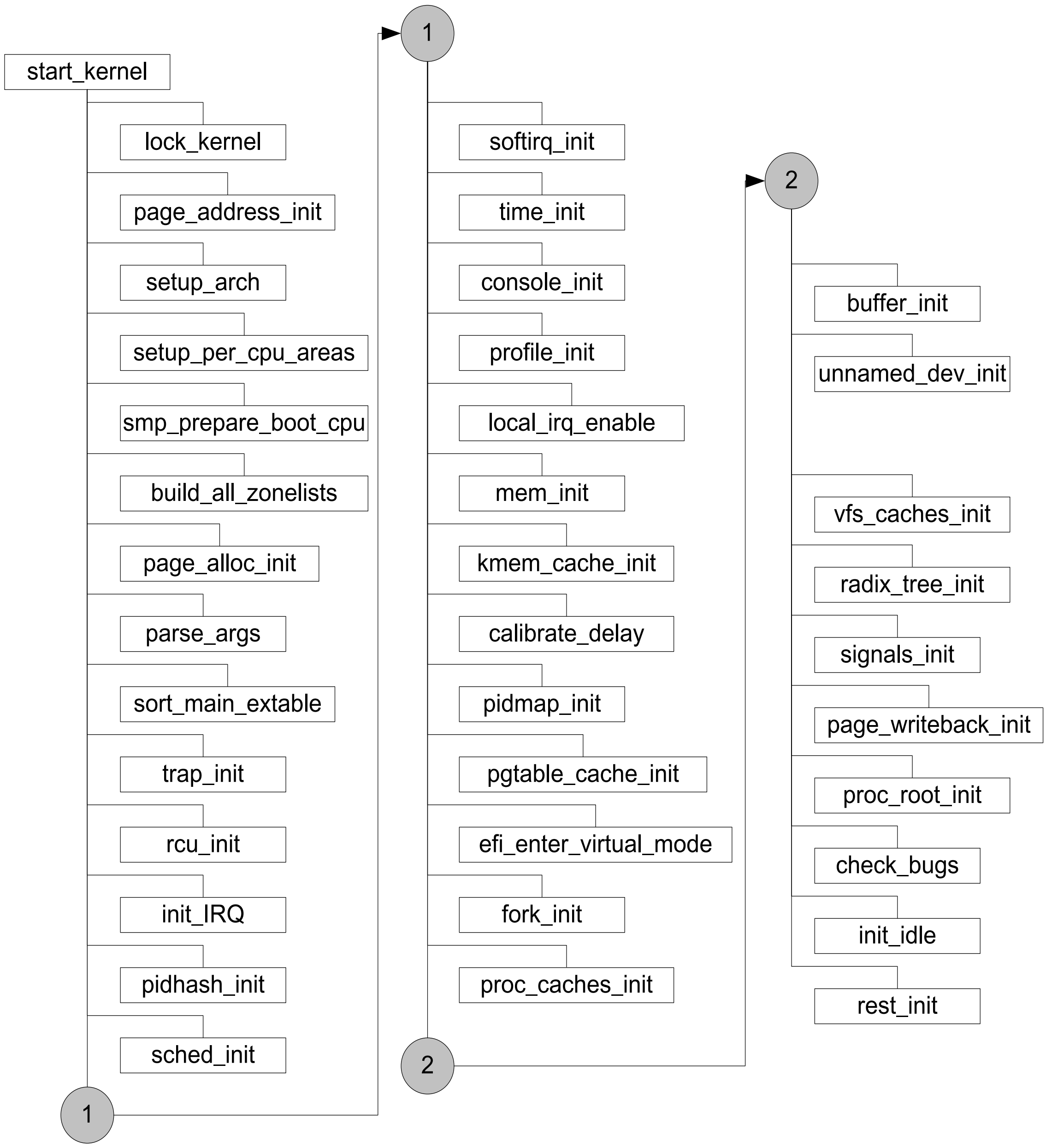
2.1　系统初始化流程简介

Linux 系统的启动，指的是从系统加电后直至系统控制台显示“login:”登录提示符为止的系统运行阶段。与这部分动作密切相关的代码主要是：

- 四个汇编程序：bootsect.S　setup.S　head.S　 entry.S
- init 目录下的 main.c 文件

本节介绍的程序流程不多，这是由于汇编代码在协议栈中没有太多关系，我们可以直接分析 C 语言编写的初始化代码。另一方面，研究汇编不是我的特长，而且也没有必要。

下面看 init/main.c 中的 start_kernel 函数的分析（这是 2.6.5 内核的启动流程图，自从依据此版本画出这副：



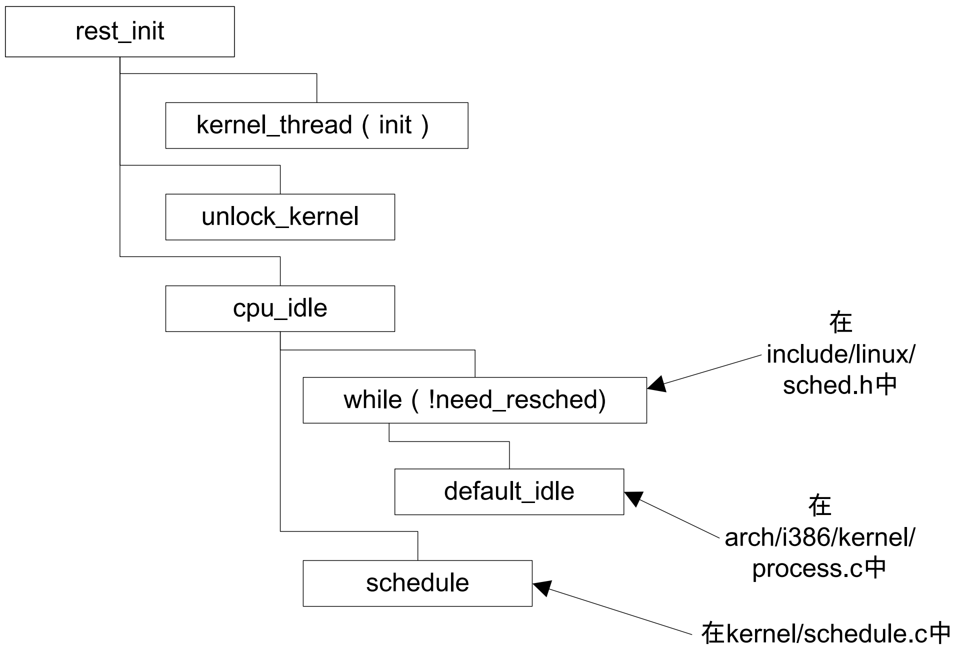
图表 2-2系统启动函数序列图

在系统启动过程中，我们要关注这几个方面：

1. 中断系统及调度系统
2. 文件系统的初始化
3. 设备管理系统的初始化
4. 网络协议的初始化

这 4 点分别是本章的 4 个小节，但在介绍这 4 个方面的内容之前必须提前介绍关于 ELF 文件格式的基本常识。最后一节会讲解网络协议栈本身的初始化。为什么要做这样的安排？因为网络系统本身关系到设备、文件系统、任务调度等方面，如果这些具体的问题没有搞清楚，那么理解协议栈本身是比较困难的。而关于 ELF 文件格式的内容则是更加重要，因为仅仅理解代码并不困难，而要理解编译器在生成内核的过程中做了什么事是区别内核与普通可执行文件的关键。

不过，在上图的函数中没有发现与网络相关的，那么它隐藏在哪呢？再看看 rest_init 函数吧。下面是分析 init/main.c 中 rest_init 函数:



图表 2-3rest_init 函数调用树

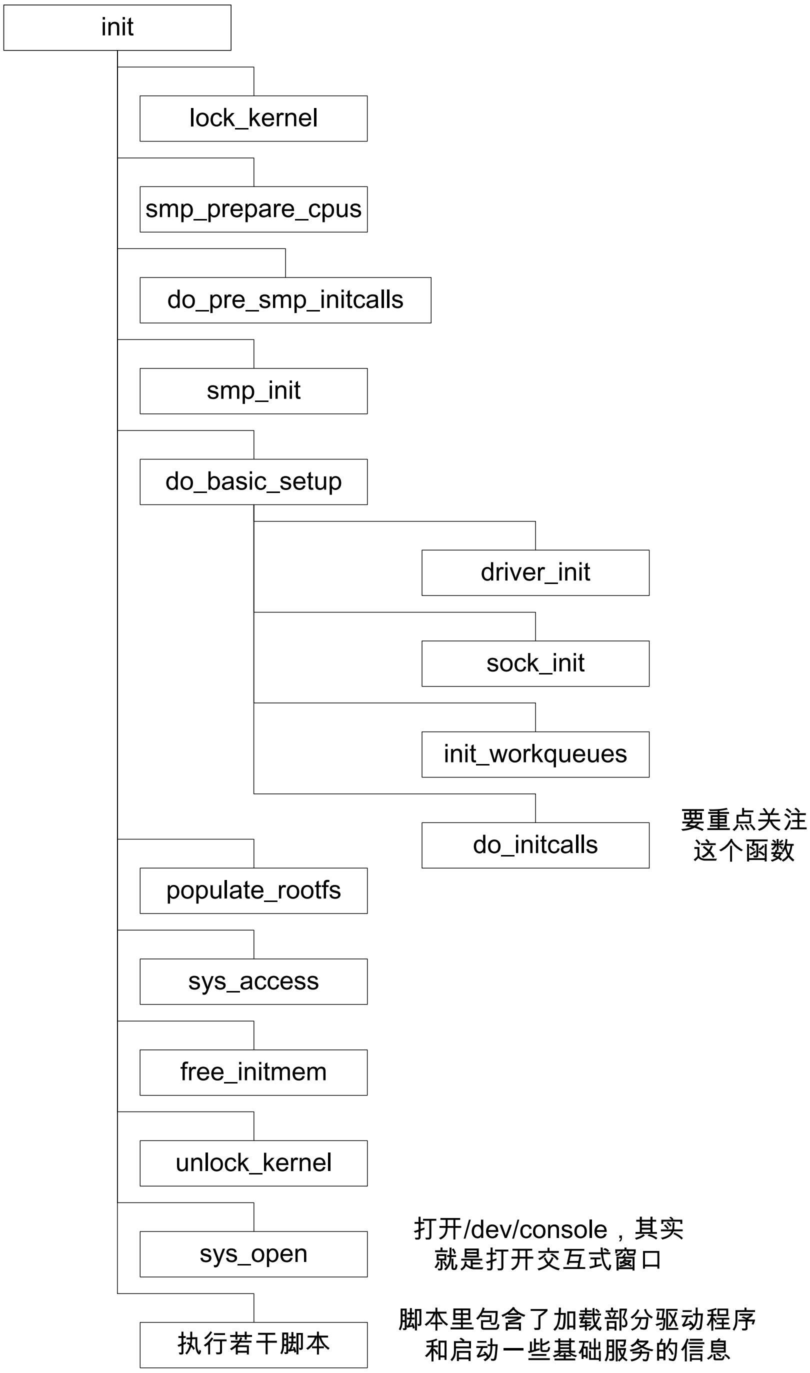
在此函数中，还是没看到关于网络初始化相关代码，但你看到那个 kernel_thread 函数了吗？那个函数创建一个内核线程，原型如下：

int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)

此函数定义在 arch/i386/kernel/process.c 中，它利用 linux/i386 的 do_fork 函数创建一个新的内核态线程，Linux 的内核线程是没有虚拟存储空间的进程，它们运行在内核中，直接使用物理地址空间。

在这里，kernel_thread 创建的新的内核线程是 init，然后返回，执行 unlock_kernel（与 start_kernel 中的 lock_kernel 对应），接着执行 cpu_idle()，这实际是执行初始化主线程的归宿：它观察自己是否处于 TIF_NEED_RESCHED——在 need_resched 实现，如果不是，就让自己睡眠，否则完成 schedule（）函数。TIF 即 Thread Information Flag 的意思。

现在注意力转到 init 函数。Linux 内核代码中有多个模块定义了 init 函数，这是因为许多设备驱动程序自己实现了名字叫做 init 的函数，大多数都是我们不关心的内容，所以必须找到正确的那个。众里寻它千百度，默然回首，它居然就在 init/main.c 中。



图表 2-4init 函数调用关系树

init 线程好像调用了许多函数，但终于只在 do_basic_setup 中看到了关于和网络有关系的初始化了——sock_init（），但是我们跟踪进这个函数，发现它只是为网络创建了执行环境，并为协议栈申请了内存空间。我们将在“***网络系统初始化***”一节详细分析此函数。但是协议栈本身在哪被初始化呢？

在这样的函数调用树中找不到直接关于网络的初始化函数，它们在哪？我们将注意力移到 do_initcalls 这个函数，在此函数定义的 c 文件中，有这样两个变量__initcall_start 和__initcall_end，它们是如下定义和的：

```
1.  extern initcall_t __initcall_start, __initcall_end;
2.
3.  static void __init do_initcalls(void)
4.  {
5.  initcall_t *call;
6.  int count = preempt_count();
   从__initcall_start这个变量开始遍历，直到遇到__initcall_end这个变量
7.  for (call = &__initcall_start; call < &__initcall_end; call++)
8.  {
9.      char *msg;
10.         ……
  调用初始化函数，我们目前是无法从代码上直接看到call是什么函数
11.     (*call)();
12.
13.     msg = NULL;
14.     ......
15. }
16.
17. /* Make sure there is no pending stuff from the initcall sequence */
18. flush_scheduled_work();
19. }
```

代码段 2-1do_initcalls 函数

我们用C源码分析工具没有观察到**__initcall_start, __initcall_end**是在哪个C文件和H文件中被定义的。难道没有定义它们编译器就让这种“错误”蒙混过关了吗？我们注意到call变量是initcall_t类型的，这是一种什么样的类型呢？只好到跟**initcall_t**相关的文件中去找。于是在include/linux目录下发现init.h文件定义了这个类型变量，下面章节我们会介绍具体的定义，为了介绍为什么要定义这样一种类型，那我们必须得知道一个不得不说的话题：ELF文件格式。下面的章节详细讲解了关于这方面的知识，对于了解操作系统工作也是很好的教材。

2.2 内核文件解读

为了解决 initcall_t 到底是什么变量类型则必须提及 C 语言中一个比较少见的内容——可执行文件格式，只有了解了这种文件格式才能具体知道 initcall_t 的意义。下面我们就开始了。

2.2.1 ELF 文件格式

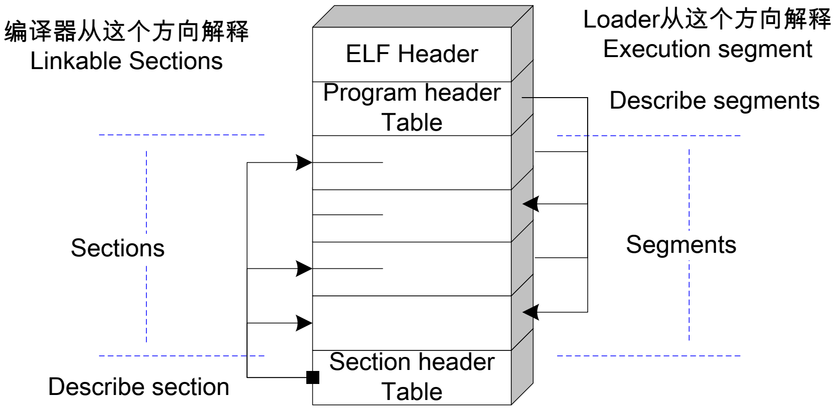
ELF 是*nix 系统上可执行文件的标准格式，它取代了 out 格式的可执行文件，原因在于它的可扩展性。

ELF 格式的可执行文件可有多个 section。DWARF（Debugging With Attribute Record Format）是经常碰到的名词，它在 ELF 格式的可执行文件中。

ELF文件有三种不同的形式：

- Relocatable：由编译器和汇编器生成，由 linker 处理它。
- Executable：所有的重定位和符号解析都完成了，也许共享库的符号要在运行时刻解析。
- Shared Object：包含 linker 需要的符号信息和运行时刻所需的代码。

ELF 文件有双重性质：一方面，编译器、汇编器、连接器都把它看作是逻辑段（sections）的集合，另一方面 loader 把它看作段（segments）的集合。Section 是给 linker 做进一步处理的，而 segments 是被映射到内存中去的。（中文里面 section 可以叫做节也可以叫段，而 segment 亦然，为避免歧义，这里坚持用英文表示）一个 segment 可以由几个 sections 组成。为了定位不同 segment/section，可执行文件用一个 table 来记录各个 segment/section 的位置和描述。Relocatable 有 section table，Executable 有 program header table。而 Shared Object 两者都有。

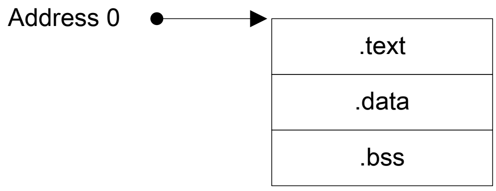


图表 2-5ELF 文件格式

上图演示了从不同的角度来解释 segment 和 section 的不同。

当 as 生成一个目标文件时,它假设程序段是从地址 0 开始,ld 则把最后的地址赋给这个程序段.以至于不同的程序段不会相互复盖。ld 把程序移动到各自的运行时地址,指定 section 的运行时地址叫重定位(relocation)。

as 输出的目标文件至上有三个 section,任何一个都有可能为空,它们是 text,data,bss 段。你可以不写诸如.text 或.data 段,但目标文件中还是存在这些段,只不过是空的，在目标文件中段是如下排列:



图表 2-6普通的 ELF 段排列

为了让 ld 能正确重定位各段，as 生成一些重定位所需的信息。

实际上 as 用的每一个地址都是以这样的形式：(section)+(offset into section)表示，ld 把所有相同的 section 放到连续的地址里。你还可以用 subsections 把一个大的 section 分成多个小的 section。可以用标号来区分，这里就不细说了。

普通情况下 ld 处理四种段：

named section：

text section

data section

这两个段放着你的程序，它们是分开的但是却是相等的段。只是在运行的时刻，text 段不能被改变。

bss section

absolute section

这个段的 0 地址总是被重定位到运行地址 0

undefined section

用来放置不在前面几个段里的数据。

对于一个在 text、data、bss 中的符号而言，它的值就是从段首到它的偏移，于是，当 ld 在连接各段时就改变了 label 的值。

对于没有定义（undefined）的值，ld 尽量从外部其他文件引入并确定其值。

不同的 sections 的含义（大家都知道的我就不说了）：

- .dynamic：该 section 保存着动态连接的信息。
- .dtnstr：保存动态连接时需要的字符串。
- .dynsym：保存动态符号表如“symbol table”的描述。
- .interp：保存程序的解释程序(interperter)的路径。

- .line：包含编辑字符的行数信息，它描述源代码与机器代码之间的对应关系。
- .rel<name>和.rela<name>：保存重定位的信息。
- .rodata 和.rodata1：保存只读数据，在进程映像中构造不可写的段。

前缀是点(.)的 section 名是系统保留的。

2.2.2 Link Scripts 知识

我们通常有一个疑问，为什么我们编出来的代码肯定是在用户地址空间运行，而内核编出来的代码却一定是运行在内核空间？如果我们把目光仅仅盯在 C 文件或 h 文件甚至 Makefile，估计想破脑袋也不知道为什么会这样。其实我们经常忽视了链接器的作用。不能简单地认为链接器仅仅完成将各 obj 文件拼在一起的任务，而且它还指定每个段被装入内存的真正地址，没错，是装入！

链接器（Linker）其实有自己的一套语言规范，其目的是描述输入文件中的 sections 是如何映射到输出文件中，并控制输出文件的内存排列。如果你从来没有看到过 ld script，那么请用 ld -verbose 查看输出结果，那就是 ld script。只是它是内置在链接器中，而且 ld 就是使用这个缺省的 script 去生成输出我们平时应用程序 obj，所以如果是用缺省的 ld script 生成内核，那么它肯定也只能跑在用户空间。

我们已经知道每一个目标文件有一个 sections 的列表，在输入文件中，是 input section，在输出文件中叫 output section。每个 section 有名字和大小。大多数 section 有相关的数据块，就是 section contents。一个 section 可以是标记为 loadable，意味着输出文件在运行时可以把这一 section 装入内存。没有内容的 section 可以叫 allocatable，表示这块区域放在内存的某个地方，但没有什么特殊的东西放在里面（一般都是被初始化为 0），一个既不是 loadable 也不是 allocatable 的 section 一般是包含一堆调试信息。

每一个 loadable 和 allocatable 的 section 有两个地址。第一个是 VMA，即虚存地址。这是输出文件运行时的地址。第二个是 LMA，即装入内存地址。这是 section 被装入的地址。在多数情况下，这两个地址是相同的。他们不同的例子是：当数据 section 被装入 ROM，当程序开始执行时被复制到 RAM（这个技术通常用来初始化基于 ROM 系统的全局量）。

你可以用 dumpobj -h 去查看目标文件的 section 信息。每个目标文件有符号表。每个符号有一个名字，且每个有定义的符号有一个地址及其他信息。你可以用 nm 查看符号表信息，也可以用 objdump -t 命令查看。

最简单的 script 只有一个 SECTIONS 命令。它描述输出文件的内存排列。

例子：

```
SECTIONS
{
    .=0x10000;     /*代码被装入到此地址*/
    .text SIZEOF_HEADERS:{
*(.init)
*(.text)
*(.fini)
}
    .=0x8000000; /*数据被装入到此地址*/
    .data:{*(.data)}
    .bss:{*(.bss)}
}
```

在上例中，第 3 行的'.'是一个特殊符号，用来做定位计数器。它根据输出段的大小增长。在 SECTIONS 开始时它等于 0。

‘*'是一个通配符，匹配所有的文件，表达式"*(.text)"表示所有输入文件的".text"段。输出文件的.text 段包含所有输入文件的.init 和.text 及.fini。

在 linker 放置".data"后，定位计数器的值等于 0x8000000 加上".data"的大小。然后 linker 会把.bss 段放在.data 之后。注意：linker 可能会在.data 和.bss 段之间划出一个 gap。

程序中执行的第一条指令叫 entry point，可以用 ENTRY 指定入口点。如 ENTRY(symbol)。linker 有几种方法设置入口点：

1。在命令行中输入-e entry
2。在 linker script 文件中指定 ENTRY(symbol)
3。如果定义了 start，则 start 的值就是入口点

4。.text 的第一个字节
5。地址 0

在一些目标文件里，公共符号不属于某个特别的段，linker 认为它们属于一个叫 COMMON 的段，大多数情况，输入文件里的公共符号被放在输出文件的.bss 段中，如
.bss {*(.bss) *(COMMON)}

输出段属性
linker 一般在 input section 的基础上设置 output section 的属性。
可以使用 AT 命令改变地址值。
覆盖命令提供了一个简单的方法把不同的 section 装入单一内存镜像中，但在执行的时候是从同一的地址开始执行。
PROVIDE 输出符号以便让 linker 能在解析过程中用到。用法是 PROVIDE(symbol = express)。例如

```
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

如果程序里定义了_etext，则 linker 会报错：多个_etext 的定义。但如果程序定义了 etext，则编译器默认使用程序里的 etext 定义；如果程序没有定义 etext 但却用到了 etext，则 linker 使用 link script 中的定义。

2.2.3 Linux 内核镜像研究

下面我们就拿 Linux 内核源代码作为复习以上的例子。先回顾 include/linux 目录下这么一个 init.h 文件，它不仅定义了 initcall_t 类型变量，还定义了一些常规 C 语言编程中未见过得类型：

```
        用法:
        对于函数，应该在函数名之前加一个__init，如下:
1.   /* static void __init initme(int x, int y)
2.   * {
3.   *    extern int z; z = x * y;
4.   * }
        如果函数在其他地方有原型，那么你可以在括号和分号之间加__init，如下:
5.   * extern int initialize_foobar_device(int, int, int) __init;
6.   *
        对于初始化的数据，应该在变量和等号之间加一个__initdata，如下:
7.   * static int init_variable __initdata = 0;
8.   * static char linux_logo[] __initdata = { 0x32, 0x36, ... };
       记住不要在文件范围以外初始化数据,比如在函数中，否则 gcc 会把这些数据放在 bss 段中而不是 init 段中。
9.
        而且要注意：这些数据不能是 const 类型
10. */
11.
12. /* 这里要碰到一个__attribute__关键字，这是告诉编译器要做一些特殊的操作。在这里，它和__section__
    共同指示凡是被它们修饰的函数或变量应该放在特殊的 section 中,不能任由编译器自己决定这些函数放在哪 */
13. #define __init      __attribute__ ((__section__ (".init.text")))
14. #define __initdata  __attribute__ ((__section__ (".init.data")))
15. #define __exitdata  __attribute__ ((__section__(".exit.data")))
16. #define __exit_call __attribute_used__ __attribute__ ((__section__
    (".exitcall.exit")))
17.
18. #define __sched     __attribute__((__section__(".sched.text")))
19.
20. #ifdef MODULE
21.   #define __exit    __attribute__ ((__section__(".exit.text")))
```

```
22. #else
23.   #define __exit   __attribute_used__ __attribute__ ((__section__(".exit.text")))
24. #endif
25.
26. /* For assembly routines */
27. #define __INIT      .section    ".init.text","ax"
28. #define __FINIT     .previous
29. #define __INITDATA  .section    ".init.data","aw"
```

系统中没有定义__ASSEMBLY__宏，所以会编译下面的代码

```
30. #ifndef __ASSEMBLY__
```

这就是前面我们提到的函数类型定义，它是一个参数为空，返回值为 int 类型的函数，还有 exitcall_t 类型函数类型定义，参数为空，返回值也为空。它们使用了标准的 typedef 用法，没有什么可以研究的

```
31.   typedef int (*initcall_t)(void);
32.   typedef void (*exitcall_t)(void);
33.
34.   extern initcall_t __con_initcall_start, __con_initcall_end;
35.   extern initcall_t __security_initcall_start, __security_initcall_end;
36. #endif
37.
38. #ifndef MODULE
39.
40.   #ifndef __ASSEMBLY__
41.
42.        /* initcalls are now grouped by functionality into separate
43.         * subsections. Ordering inside the subsections is determined
44.         * by link order.
45.         * For backwards compatibility, initcall() puts the call in
46.         * the device init subsection.
47.         */
48.
49.        #define __define_initcall(level,fn) \
50.         static initcall_t __initcall_##fn __attribute_used__ \
51.         __attribute__((__section__(".initcall" level ".init"))) = fn
52.
53.        #define core_initcall(fn)       __define_initcall("1",fn)
54.        #define postcore_initcall(fn)   __define_initcall("2",fn)
55.        #define arch_initcall(fn)       __define_initcall("3",fn)
56.        #define subsys_initcall(fn)     __define_initcall("4",fn)
57.        #define fs_initcall(fn)         __define_initcall("5",fn)
58.        #define device_initcall(fn)     __define_initcall("6",fn)
59.        #define late_initcall(fn)       __define_initcall("7",fn)
```

我们常见的__initcall 宏实际指的是 device_initcall

```
60.        #define __initcall(fn) device_initcall(fn)
61.
62.        #define __exitcall(fn) \
63.         static exitcall_t __exitcall_##fn __exit_call = fn
64.
65.        #define console_initcall(fn) \
66.         static initcall_t __initcall_##fn \
67.         __attribute_used__ __attribute__((__section__(".con_initcall.init")))=fn
68.
69.        ......
70.        struct obs_kernel_param {
71.         const char *str;
72.         int (*setup_func)(char *);
73.        };
74.
75.     ......
76.    #endif /* __ASSEMBLY__ */
77.
```

module_init() – 声明驱动程序初始化入口函数的宏，它实际就是上面提到的 device_initcall
@x: 此参数是内核 boot 时或将模块插入内核时将要调用的驱动程序函数
凡是被 module_init()“修饰”过的函数只能在两种 情况下被调用：一种是被 do_initcalls 调用，一种是在模块插入到系统中时被调用（如果它是模块方式）。每个模块只有一个被 module_init 修饰的函数入口

```
78. #define module_init(x)  __initcall(x);
79.
```

```
    module_exit() – 声明驱动程序退出函数的宏，
    @x: 此参数是驱动程序被卸载时内核时将要调用的函数
    module_exit() 会封装驱动程序的 clean-up 代码，如果驱动程序静态编译到内核中，这个宏没有意义。每个
    模块也只有一个被 module_exit 修饰的函数
80. #define module_exit(x)  __exitcall(x);
81.
82.  #else /* MODULE */
83.
84. /* Don't use these in modules, but some people do... */
85. #define core_initcall(fn)       module_init(fn)
86. #define postcore_initcall(fn)   module_init(fn)
87. #define arch_initcall(fn)       module_init(fn)
88. #define subsys_initcall(fn)     module_init(fn)
89. #define fs_initcall(fn)         module_init(fn)
90. #define device_initcall(fn)     module_init(fn)
91. #define late_initcall(fn)       module_init(fn)
92.
93. #define security_initcall(fn)       module_init(fn)
94.
95. /* These macros create a dummy inline: gcc 2.9x does not count alias
96.  as usage, hence the `unused function' warning when __init functions
97.  are declared static. We use the dummy __*_module_inline functions
98.  both to kill the warning and check the type of the init/cleanup
99.  function. */
100.
101.    /* Each module must use one module_init(), or one no_module_init */
102.    #define module_init(initfn)                \
103.        static inline initcall_t __inittest(void)      \
104.        { return initfn; }                 \
105.        int init_module(void) __attribute__((alias(#initfn)));
106.
107.    /* This is only required if you want to be unloadable. */
108.    #define module_exit(exitfn)                \
109.        static inline exitcall_t __exittest(void)      \
110.        { return exitfn; }                 \
111.        void cleanup_module(void) __attribute__((alias(#exitfn)));
112.
113.    #define __setup_param(str, unique_id, fn)   /* nothing */
114.    #define __setup_null_param(str, unique_id) /* nothing */
115.    #define __setup(str, func)          /* nothing */
116.    #define __obsolete_setup(str)           /* nothing */
117. #endif
118.
119. /* This means "can be init if no module support, otherwise module load
120.    may call it." */
121. #ifdef CONFIG_MODULES
122.   #define __init_or_module
123.   #define __initdata_or_module
124. #else
125.   #define __init_or_module __init
126.   #define __initdata_or_module __initdata
127. #endif /*CONFIG_MODULES*/
128.
129. #ifdef CONFIG_HOTPLUG
130.   #define __devinit
131.   #define __devinitdata
132.   #define __devexit
133.   #define __devexitdata
134. #else
135.   #define __devinit __init
136.   #define __devinitdata __initdata
137.   #define __devexit __exit
138.   #define __devexitdata __exitdata
139. #endif
140.
141. /* Functions marked as __devexit may be discarded at kernel link time, depending
142.    on config options.  Newer versions of binutils detect references from
```

```
143.   retained sections to discarded sections and flag an error.  Pointers to
144.   __devexit functions must use __devexit_p(function_name), the wrapper will
145.   insert either the function_name or NULL, depending on the config options.
146.  */
147. #if defined(MODULE) || defined(CONFIG_HOTPLUG)
148.   #define __devexit_p(x) x
149. #else
150.   #define __devexit_p(x) NULL
151. #endif
152.
153. #ifdef MODULE
154.   #define __exit_p(x) x
155. #else
156.   #define __exit_p(x) NULL
157. #endif
```

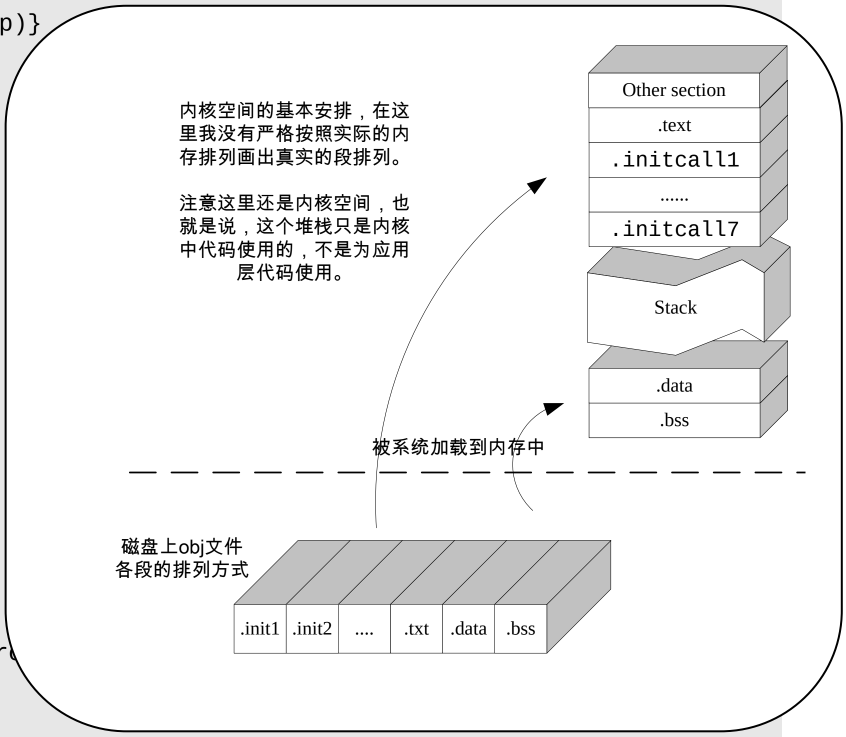
代码段 2-2init.h

这个文件有一些关于 section 的定义比如在第 16 行和 30 行，所以本质上 initcall_t 及其他类似的类型变量实际是宏，通过对这个文件的宏替换可以大概了解这些宏的含义。例如，代码中如果含有__init XXX（）这么一个函数定义，你就知道那个 XXX 函数属于初始化的时候就被调用的，它被放在.init.text 节中。

前面说到 ELF 文件格式时曾说了 Link Script 会把特定类型的段放在特定位置让 loader 装入到特定的内存地址，那么对于被 init 等宏修饰的函数及全局变量肯定被放在了特定位置，但如何让 basic_init 函数去调用它，我们还得再研究 arch/i386/kernel 目录下的 vmlinux.lds.S 文件，它就是使内核成为内核的 ld script。在这个文件中便定义了__initcall_start 和__initcall_end。记住了，不止是 C 文件和 H 文件可以定义变量。而且也不是象 C 语言那样定义一个变量还要指定类型，编译器有足够的智商把这个文件中的变量定义为需要的类型，一般情况下会定义为整型变量。

```
    正是这个 ld script 创建了 Linux 内核
1.
2.  #include <asm-generic/vmlinux.lds.h>
3.  #include <asm/thread_info.h>
4.
5.  #include <linux/config.h>
6.  #include <asm/page.h>
7.  #include <asm/asm_offsets.h>
8.
9.  OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
10. OUTPUT_ARCH(i386)
11. ENTRY(startup_32)
12. jiffies = jiffies_64;
13. SECTIONS
14. {
15.   . = __PAGE_OFFSET + 0x100000;
16.   /* 代码段必定是只读的 */
17.   _text = .;        /* Text and read-only data */
18.   .text : {
19. *(.text)
20. SCHED_TEXT
21. *(.fixup)
22. *(.gnu.warning)
23. } = 0x9090
24.
25.   .entry.text : { *(.entry.text) }
26.
27.   _etext = .;           /* End of text section */
28.
   ......  //只读数据段
29.   RODATA
30.
31.   /* 可写数据段 */
32.   .data : {         /* Data */
33. *(.data)
34. CONSTRUCTORS
35. }
```

```
    ......
36.   . = ALIGN(32);
37.   .data.cacheline_aligned : { *(.data.cacheline_aligned) }
38.
39.   _edata = .;           /* 数据 section 在此结束*/
40.
41.   . = ALIGN(THREAD_SIZE);   /* init_task */
42.   .data.init_task : { *(.data.init_task) }
43.
44.   /* will be freed after init */
45.   . = ALIGN(PAGE_SIZE_asm);     /* 下面是初始化代码和数据 */
46.   __init_begin = .;
47.   .init.text : {
48. _sinittext = .;
49. *(.init.text)
50. _einittext = .;
51.   }
52.   .init.data : { *(.init.data) }
53.   . = ALIGN(16);
54.   __setup_start = .;
55.   .init.setup : { *(.init.setup)}
56.   __setup_end = .;
57.   __start___param = .;
58.   __param : { *(__param) }
59.   __stop___param = .;
     这就是我们日思夜想的那两个全局变量
60.   __initcall_start = .;
61.   .initcall.init : {
62. *(.initcall1.init)
63. *(.initcall2.init)
64. *(.initcall3.init)
65. *(.initcall4.init)
66. *(.initcall5.init)
67. *(.initcall6.init)
68. *(.initcall7.init)
69.   }
70.   __initcall_end = .;
71.
    ......
72.   .exit.text:{*(.exit.text) }
73.   .exit.data:{*(.exit.data) }
74.   . = ALIGN(PAGE_SIZE_asm);
75.   __per_cpu_start = .;
76.   .data.percpu  : { *(.data.perc
77.   __per_cpu_end = .;
78.   . = ALIGN(PAGE_SIZE_asm);
79.   __init_end = .;
     /* 以上的内存段在初始化结束之后（即这些代码被执行一次以后）就被释放出来 */
80. ......
     这里存放未初始化的数据
81.   __bss_start = .;      /* BSS */
82.   .bss : {
83. *(.bss.page_aligned)
84. *(.bss)
85.   }
86.   . = ALIGN(4);
87.   __bss_stop = .;
88.
89.   _end = . ;
90.
91.   /* This is where the kernel creates the early boot page tables */
92.   . = ALIGN(4096);
93.   pg0 = .;
94.
95.   /* Sections to be discarded */
96.   /DISCARD/ : {
97. *(.exitcall.exit)
```



```
98. }
99.
100.      /* Stabs debugging sections.  */
101.      /* 省略 */
102.      ......
103.    }
```

代码段 2-3Linux 内核 ld scripts

看到了吗？当编译器编译整个源代码的时候，它会把所有定义为__init 的函数放在以__initcall_start开始、以__initcall_end 结束的节中，在 basic_init 中会逐个的调用该节里所有的函数。不信？让我们编译完内核，我们在源文件根目录使用 objdump –t vmlinux |grep _initcall 输出信息，就看到我们想看到的东西（下面是经过处理的符号信息）：

```
1.  c12946d4    l   O   4   __initcall_cpufreq_tsc 第一个被初始化的函数，它的地址正好是
    __initcall_start 的地址
2.  ......
3.  c12946e4    l   O   4   __initcall_ksysfs_init
4.  ......
5.  c12946fc    l   O   4   __initcall_sock_init        网络基础设施比如内存系统的初始化
6.  c1294700    l   O   4   __initcall_netlink_proto_init
7.  ......
8.  c1294770    l   O   4   __initcall_proto_init
9.  c1294774    l   O   4   __initcall_net_dev_init     网络设备的初始化
10. c1294778    l   O   4   __initcall_wireless_nlevent_init
11. c129477c    l   O   4   __initcall_pktsched_init
12. .......
13. c129478c    l   O   4   __initcall_init_pipe_fs      管道系统的初始化
14. c1294790    l   O   4   __initcall_chr_dev_init      字符设备的初始化
15. .......
16. c12947a4    l   O   4   __initcall_inet_init         网络系统的初始化
17. c12947a8    l   O   4   __initcall_time_init_device
18. ......
19. c12947b0    l   O   4   __initcall_i8259A_init_sysfs 中断控制器的初始化
20. ......
21. c12947e8    l   O   4   __initcall_init_posix_timers
22. c12947ec    l   O   4   __initcall_init_posix_cpu_timers
23. c12947f0    l   O   4   __initcall_init_clocksource_sysfs
24. c12947f4    l   O   4   __initcall_init_jiffies_clocksource
25. ......
26. c1294848    l   O   4   __initcall_eventpoll_init
27. ......
28. c1294864    l   O   4   __initcall_init_ext2_fs      EXT2 文件系统的初始化
29. ......
30. c1294888    l   O   4   __initcall_init             这不是 init/main.c 文件中的初始化函数
31. ......
32. c12948a8    l   O   4   __initcall_pci_init          PCI 系统总线的初始化
33. c12948ac    l   O   4   __initcall_pci_sysfs_init
34. c12948b0    l   O   4   __initcall_pci_proc_init
35. ......
36. c1294918    l   O   4   __initcall_topology_sysfs_init
37. c129491c    l   O   4   __initcall_rd_init
38. c1294920    l   O   4   __initcall_net_olddevs_init  老实的网络设备驱动程序的初始化
39. ......
40. c1294974    l   O   4   __initcall_generic_ide_init  IDE 硬盘设备的初始化
41. ......
42. c12949b4    l   O   4   __initcall_mousedev_init      鼠标设备的初始化
43. ......
44. c12949cc    l   O   4   __initcall_flow_cache_init
45. c12949d0    l   O   4   __initcall_blackhole_module_init
46. c12949d4    l   O   4   __initcall_init_syncookies
47. ....
48. c12949e0    l   O   4   __initcall_af_unix_init
49. c12949e4    l   O   4   __initcall_packet_init
50. ......
```

```
51. c1294a30    l   O   4   __initcall_net_random_reseed最后一个被调用的初始化函数，它的地址
    正好是__initcall_end 的前面 4 个字节
52.
53. c12946d4    g   "*ABS*  00000000 __initcall_start"
54. c1294a34    g   "*ABS*  00000000 __initcall_end"
55. ......
    下面是一些处于初始化 section 的函数
56. c03ce000    l   d   ".data.init_task    00000000"
57. c03d0000    l   d   ".init.text 00000000"
58. c03ece80    l   d   ".init.data 00000000"
59. c03f57e0    l   d   ".init.setup    00000000"
60. c03f5c78    l   d   ".initcall.init 00000000"
61. c03f5f84    l   d   ".con_initcall.init 00000000"
62. c03f5f90    l   d   ".security_initcall.init    00000000"
63. ......
64. c03d0510    l   F   ".init.text 00000025 init_setup"
65. c03f5840    l   O   ".init.setup    0000000c __setup_init_setup"
66. c0100220    l   F   ".text  0000002a rest_init"
67. c0100290    l   F   ".text  00000151 init"
68. c03d0570    l   F   ".init.text 00000091 do_early_param"
69. c03d0840    l   F   ".init.text 000000bc do_initcalls"
70. c03d0900    l   F   ".init.text 0000001f do_basic_setup"
71. c0100260    l   F   ".text  00000029 run_init_process"
72. c010b090    l   F   ".text  00000127 init_intel"
73. c01c8340    g   F   ".text  00000034 kobject_init"
74. c0248e90    g   F   ".text  00000089 device_initialize"
75. c02f09c0    g   F   ".text  0000011c tcp_select_initial_window"
76. c03e93f0    g   F   ".init.text 00000053 loopback_init"
77. c01254d0    g   F   ".text  00000015 init_timer"
78. c03d7aa0    g   F   ".init.text 00000062 init_IRQ"
79. c03eb650    g   F   ".init.text 0000006d skb_init"
80. c03df270    g   F   ".init.text 000003eb kmem_cache_init"
81. c03ecb00    g   F   ".init.text 00000090 ip_misc_proc_init"
82. c031f150    g   F   ".text  00000033 klist_init"
83. a5808bbf    g       "*ABS*  00000000 tasklet_init"
84. c03dcde0    g   F   ".init.text 000000fd proc_caches_init"
85. c0307050    g   F   ".text  000000c9 ip_mc_init_dev"
86. c02e57c0    g   F   ".text  0000007a inet_csk_init_xmit_timers"
87. c039ba80    g   O   ".data  00000048 tcp_init_congestion_ops"
88. c03dd590    g   F   ".init.text 00000027 init_timers"
89. c012d460    g   F   ".text  0000001f init_workqueues"
90. c03dd410    g   F   ".init.text 0000001f softirq_init"
91. c03ec400    g   F   ".init.text 0000000a tcp4_proc_init"
92. c03eb990    g   F   ".init.text 000000b9 rtnetlink_init"
93. c0121ef0    g   F   ".text  0000001b tasklet_init"
94. c0281d20    g   F   ".text  00000065 ide_init_disk"
95. c03dd860    g   F   ".init.text 0000000f sort_main_extable"
96. c03ec0d0    g   F   ".init.text 00000329 tcp_init"
97. 45e55588    g       "*ABS*  00000000 _inet_csk_init_xmit_timers"
98. c0377b4c    g   O   ".data  00000014 __init_timer_base"
99. c03fa000    g       "*ABS*  00000000 __init_end"
100.    c03dd430    g   F   ".init.text 0000002e spawn_ksoftirqd"
101.    c03e8bf0    g   F   ".init.text 00000032 classes_init"
102.    c03f2660    g       ".init.data 00000000 vsyscall_int80_end"
103.    c03ec090    g   F   ".init.text 0000000f ip_init"
104.    c02f3e50    g   F   ".text  0000001b tcp_init_xmit_timers"
105.    c03d0670    g   F   ".init.text 000001b8 start_kernel"
106.    c03dcb80    g   F   ".init.text 00000001 pre_setup_arch_hook"
107.    c03dd570    g   F   ".init.text 0000001a sysctl_init"
108.    c03ebd40    g   F   ".init.text 000002ab ip_rt_init"
109.    c03e0e10    g   F   ".init.text 000000cd ipc_init_ids"
110.    c03ecb90    g   F   ".init.text 0000000a fib_rules_init"
111.    c030d640    g   F   ".text  000000ee fib_hash_init"
112.    c03ec4d0    g   F   ".init.text 00000056 arp_init"
113.    c02ea280    g   F   ".text  0000003d tcp_init_cwnd"
114.    c03ec410    g   F   ".init.text 00000065 tcp_v4_init"
115.    c03ed05c    g   O   ".init.data 00000004 root_device_name"
```

这就是我们要找的那两个全局变量，请注意它们的地址

```
116.    c03ec5e0    g   F   ".init.text 00000040 devinet_init"
117.    ae005c8b    g       "*ABS*  00000000 _tcp_init_xmit_timers"
118.    c03eb6c0    g   F   ".init.text 00000097 netdev_boot_setup"
119.    c02cf590    g   F   ".text  0000027d neigh_table_init"
120.    9a72cf28    g       "*ABS*  00000000 _tcp_init_congestion_ops"
121.    c03dd380    g   F   ".init.text 00000047 profile_init"
122.    c03e8bb0    g   F   ".init.text 0000000a devices_init"
123.    231bbd1f    g       "*ABS*  00000000 _neigh_table_init"
124.    c03f1dd0    g       ".init.data 00000000 vsyscall_int80_start"
125.    c03e0ae0    g   F   ".init.text 0000000a init_rootfs"
126.    c01084c0    g   F   ".text  0000008a init_8259A"
127.    1ac12182    g       "*ABS*  00000000 _inode_init_once"
128.
```

代码段 2-4内核镜像输出 init 的打印

要对这份输出信息做一个说明：第一列是所有符号的装入地址；第二列表示符号的全局性和可读写性，l 表示 local，凡是用 static 修饰的都是这种属性，g 表示 global，内核源代码中全局变量属于这一类，还有用 EXPORT_SYMBOL（这个宏在 linux/module.h 中定义）修饰的变量和函数也都是属于 global。w 表示 writable；第三列表示此符号的本质是什么，O 表示是变量（一般是全局变量），F 表示函数，d 表示可执行文件中的 section；第四列和第五列比较复杂，如果符号属于 d，那么第五列表示该 section 缺省值，一般都为 0，如果属于 O 或 F，那么第四列表示符号所属的 section，第五列则表示该符号所占用的内存大小；还有一类最特殊，就是用 ABS 修饰的符号，***ABS*** 表示绝对 (absolute)，这意味着不能将该值更改为其他的连接，我们关注的__initcall_start 和__initcall_end 就属于这一类。所以 do_initcalls 就是用来调用所有使用__initcall 标记过的函数的。

系统初始化各模块有两种方式，我们一般都可以在编译内核的时候控制，这两种方式分别是：一种是嵌入内核中；另一种是以模块加载方式。前者将设备驱动模块嵌入整个 Linux 内核（vmlinux）中，系统启动的时候会从.init 代码段执行它们的初始化函数，所以我们可以在上面那个 vmlinux 文件中找到__initcall_xxx_drv_init 函数。以上就是关于这种方式的分析。后一种就是将设备驱动模块编译成独立的可执行文件，以.ko 为后缀名放在/lib/modules/2.6.xxx/kernel/目录下。当系统启动时，内核启动代码执行/etc/rc.d/rc.sysinit 教本，其中的代码会执行 sys_init_module 内核函数把它们加载到内核中。

不管是什么方式加载，每个驱动程序执行的第一行代码是以 module_init 定义的函数。比如 module_init(e100_init_module)，这里第一行被系统执行的函数是 e100_init_module。

从很多的资料上得出一个信息：凡是用__init 修饰过的函数在被调用一次后，其占用的内存区会被清除掉，以便让其它代码可以使用。我没有时间来印证，希望有读者可以给出正确的答案。

从上图看出，网络部分的初始化代码入口被我们找到了。我们可以继续往下研究了

2.3 中断及任务调度管理

Linux 书籍中常说的 BottomHalf 已然不见了，它们被转成 tasklets，这是支持 SMP 的。但其思想基本一致。

2.3.1 中断及软中断模型

我们在此不会对中断及异常的原理和机制做深入的介绍。但必须要作出一些说明，因为这是理解 Linux 内核与其它嵌入式/实时操作系统的不同，以及理解网络协议栈收报文的基础。

Linux 支持 CPU 的外部硬件中断和内部中断。严格来说，内部中断包含系统调用陷入和异常，在一般的嵌入式操作系统（比如 VxWorks）中是没有系统调用这个概念的，所以对于一直从事嵌入式软件开发的人初次进入到大型操作系统（比如 Linux 和 Windows）开发环境中，会面临内核空间与用户空间概念上的困惑。其实说到底，所谓系统调用就是软件有计划地调用 CPU 提供的特殊指令，触发 CPU 内部产生一个中断，于是完成一次核内核外运行空间的切换，具体可以参考许多书籍。而所谓异常就是软件无意的执行了一个非法指令（比如除 0）从而造成 CPU 内部引发一次中断。

外部中断特指外部设备发出的中断信号。但这几种中断的 CPU 处理过程基本相同，即：在执行完当前指令后，或在执行当前指令期间，根据中断源所提供的“中断向量”，在内存中找到相应的 ISR（中断服务例程）然后调用之。

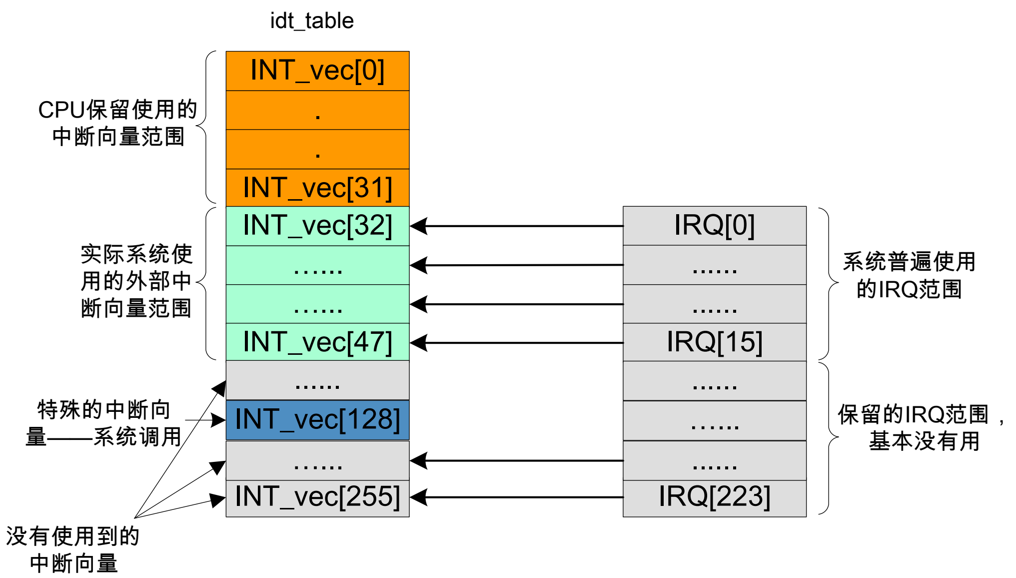
不管是内部还是外部中断，系统都会根据接收到的中断信息，查询 idt 表。idt 表依照中断源的位置

按序组成，并对应中断服务程序（以及异常处理程序）的入口地址。Linux 系统在初始化页式虚存管理的初始化以后，便调用 trap_init 和 init_IRQ 两个函数进行中断机制的初始化。我们只介绍 init_IRQ。

2.3.1.1.中断系统和软中断

中断向量？中断请求号？这是个问题。

在很多书中都提到这两个名词，但我一直没看懂这些书想表达的含义，后来有一天我发现我居然顿悟了。就让我试图给大家解释一下，看看是不是这样的：IRQ 是设备相关的号码，一般生产厂商都会使自己的设备分配到一个合适的号码。而中断向量就纯粹是操作系统中关于如何处理中断的内存组织结构，它们之间存在某种映射关系，这种关系是由 CPU 体系结构以及操作系统决定的。那么在 IA32 体系的 Linux 中，是一种直接映射的关系，所有的 IRQ 号产生的中断全部映射到从 INT_vec[32]开始的内存中。为什么要从第 32 个单元开始呢？



图表 2-7 中断向量和中断请求号之间的关系

上图中彩色部分都是系统能处理的中断，intel CPU 保留使用的中断向量是 0～32，根本不可能有哪一种设备会使用这个区域的中断向量，这一部分就是我们常说的异常处理函数，还有一个比较特殊的中断向量号 0x80（即 128）就是系统调用号，由于不可能由外部设备引发这类中断，它们就被统称为内部中断，这就是为什么要从第 32 个单元开始的原因。内核用调用 trap_init 函数挂接与之相应的中断处理函数。接着系统调用 init_IRQ 函数来初始化外部中断向量，其中断处理函数的挂接由各驱动程序自己完成。由上图可以看出中断向量和中断请求号是相关但却不是一个东西：前者是内核中存在的一块内存，专门存放中断处理函数的地址（指逻辑上，具体实现比较复杂），而后者就是一个概念，在内核中不必存在这么一种变量，它可能就是中断向量的下标。

在 2.6 的内核中，中断相关的宏已经变化了，2.4 内核中的中断概念请看《Linux 内核源代码情景分析》。在 2.6 内核的 entry.S 文件中，有一个 interrupt 的定义，它放在.data 节中，然后，在 include/asm-i386/hw_irq.h 中引用这个变量，最后在 arch/i386/kernel/i8259.c 中初始化这个变量。

下面两个代码片断是 2.4 内核关于这个 interrupt 变量的初始化：

```
1    #define IRQ(x,y) \
2    IRQ##x##y##_interrupt
3
4    #define IRQLIST_16(x) \
5    IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
6    IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
7    IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
8    IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
9
```

```
10    void (*interrupt[NR_IRQS])(void) = {
11    IRQLIST_16(0x0),
```

代码段 2-5 2.4 中断定义宏

经过编译器的预处理，interrupt 这个函数指针数组变成：

```
1    void (*interrupt[NR_IRQS])(void) = {
2    IRQ0x00_interrupt,
3    IRQ0x01_interrupt,
4    IRQ0x02_interrupt,
5    ……
6    IRQ0x0f_interrupt,
7    }
```

从代码中看出，这样的初始化不太灵活，扩展性比较差。下面给出 2.6 内核关于 interrupt 的使用方式。首先在 entry.S 中汇编代码如下：

```
1   /*
2    * Build the entry stubs and pointer table with some assembler magic.
3    */
4   .data
5   ENTRY(interrupt)
6   .text
7
    注意这里要重复 NR_IRQ 次，
8   vector=0
9   ENTRY(irq_entries_start)
10  .rept NR_IRQS
11  ALIGN

12  1:  pushl $vector-256
13  jmp common_interrupt
14  .data
15  .long 1b
16  .text
17  vector=vector+1
18  .endr
19
20  ALIGN
21  common_interrupt:
22  SAVE_ALL
23  movl %esp,%eax
24  call do_IRQ
25  jmp ret_from_intr
```

```
#ifdef CONFIG_PCI_MSI
  #define NR_IRQS FIRST_SYSTEM_VECTOR
  #define NR_IRQ_VECTORS NR_IRQS
#else
  #ifdef CONFIG_X86_IO_APIC
    #define NR_IRQS 224 PC 机一般都是这个定义
    # if (224 >= 32 * NR_CPUS)
    # define NR_IRQ_VECTORS NR_IRQS
    # else
    # define NR_IRQ_VECTORS (32 * NR_CPUS)
    # endif
  #else
    #define NR_IRQS 16
    #define NR_IRQ_VECTORS NR_IRQS
  #endif
#endif
```

在 hw_irq.h 中有这样的定义：extern void (*interrupt[NR_IRQS])(void);在此，NR_IRQS 是 224。具体的初始化如下：

```
1   void __init init_IRQ(void)
2   {
3      int i;
4
5      /* all the set up before the call gates are initialised */
6      pre_intr_init_hook();
7
8      /*
9       * 扫描整个中断向量表
10      */
11     for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
12         int vector = FIRST_EXTERNAL_VECTOR + i;
13         if (i >= NR_IRQS)
14             break;
       如果是系统调用中断号，就初始化这个中断号
15         if (vector != SYSCALL_VECTOR)
16            set_intr_gate(vector, interrupt[i]);
17     }
18     ......
19     /*
20      * Set the clock to HZ Hz, we already have a valid vector now:
21      */
22     setup_pit_timer ();
23
24     ......
25
26     irq_ctx_init(smp_processor_id());
27  }
```

代码段 2-6init_IRQ 函数

下面是关于中断上下文的一些宏，说明中断处理到达一种什么样的程度：

IRQ_INPROGRESS	1	当前还在中断上下文中
IRQ_DISABLED	2	中断被禁止
IRQ_PENDING	4	中断被挂住
IRQ_REPLAY	8	继续中断处理
IRQ_AUTODETECT	16	自动检测中断请求
IRQ_WAITING	32	对于自动检测中断，此时可能还没有看到中断到来
IRQ_LEVEL	64	使用中断优先级别——Linux 没有使用这项特性
IRQ_MASKED	128	该中断被屏蔽了，将来不希望看到
IRQ_PER_CPU	256	每个 CPU 都有一个 IRQ

在 include/linux/irq.h 文件中，有关于中断控制器的描述，中断控制器描述符. 它包含了所有低层硬件的信息。

其中一个实例是 i8259A_irq_type，它定义在 arch/i386/kernel/i8259.c 中：

```
1   static struct hw_interrupt_type
    i8259A_irq_type = {
2      "XT-PIC",
3      startup_8259A_irq,
4      shutdown_8259A_irq,
5      enable_8259A_irq,
6      disable_8259A_irq,
7      mask_and_ack_8259A,
8      end_8259A_irq,
9      NULL
10  };
```

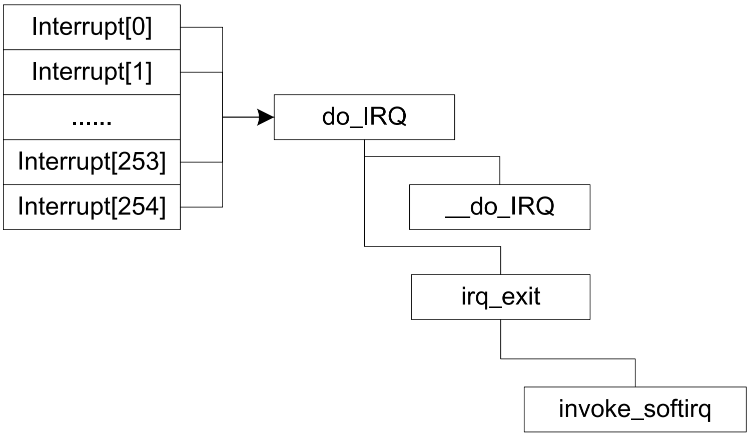
```
struct hw_interrupt_type {
    const char * typename;
    unsigned int (*startup)(unsigned int irq);
void (*shutdown)(unsigned int irq);
void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
    void (*ack)(unsigned int irq);
void (*end)(unsigned int irq);
void (*set_affinity)(uint irq, cpumask_t dest);
};
typedefstruct hw_interrupt_type  hw_irq_controller;
```

下面这个数组的定义利用了 GCC 编译器的特点，只定义了一个单元的值，然后使用[0 ... NR_IRQS-1]使整个数组的值都初始化为同样的值：

```
1   /*
2    * Controller mappings for all
    interrupt sources:
3    */
4   irq_desc_t irq_desc[NR_IRQS]
    __cacheline_aligned = {
5   [0 ... NR_IRQS-1] = {
6       .handler = &no_irq_type,
7       .lock = SPIN_LOCK_UNLOCKED
8   }
9   };
```

```
typedef struct irq_desc {
    unsigned int status;        /* IRQ 状态 */
    hw_irq_controller *handler;
    struct irqaction *action;   /* IRQ 行为列表 */
    unsigned int depth;     /*禁止嵌套 irq */
    unsigned int irq_count; /* 用来检测被阻塞的中断*/
    unsigned int irqs_unhandled;
    spinlock_t lock;
} ____cacheline_aligned irq_desc_t;;
```

现在让我们从整体上看中断和软中断的处理过程，do_IRQ 是直接被调用的：



图表 2-8 do_IRQ 函数调用树

每个外部中断都会调用 do_IRQ，此函数根据当时的 EAX 寄存器（i386 体系）值来判断当前属于哪个 IRQ 去调用__do_IRQ。

```
1   /**
2    * __do_IRQ - original all in one highlevel IRQ handler
3    * @irq:就是 do_IRQ 传入的 EAX 寄存器值
4    * @regs:指向发生中断时寄存器集合
5    *
6    * __do_IRQ 处理所有正常的设备 IRQ，它处理每一种中断类型。而特殊的 SMP CPU 有其自身的处理函数
7    **/
8   fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
9   {
10      struct irq_desc *desc = irq_desc + irq;
11      struct irqaction *action;
12      unsigned int status;
13
14      if (CHECK_IRQ_PER_CPU(desc->status)) {
15          irqreturn_t action_ret;
16          处理 IRQ 事件，见下文
17          action_ret = handle_IRQ_event(irq, regs, desc->action);
18          return 1;
19      }
20
21      spin_lock(&desc->lock);
22
23      /*
24       * REPLAY is when Linux resends an IRQ that was dropped earlier
25       * WAITING is used by probe to mark irqs that are being tested
26       */
27      status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
28      status |= IRQ_PENDING; /* we _want_ to handle it */
29
```

```
30      /*
31       * If the IRQ is disabled for whatever reason, we cannot
32       * use the action we have.
33       */
34      action = NULL;
35
36      desc->status = status;
37
38      ......
39
40      /*
41       * Edge triggered interrupts need to remember pending events.
42       * This applies to any hw interrupts that allow a second
43       * instance of the same irq to arrive while we are in do_IRQ
44       * or in the handler. But the code here only handles the _second_
45       * instance of the irq, not the third or fourth. So it is mostly
46       * useful for irq hardware that does not mask cleanly in an
47       * SMP environment.
48       */
49      for (;;) {
50          irqreturn_t action_ret;
             处理 IRQ 事件，见下文
51          action_ret = handle_IRQ_event(irq, regs, action);
52
53          desc->status &= ~IRQ_PENDING;
54      }
55      desc->status &= ~IRQ_INPROGRESS;
56
57      out:
58      ......
59
60      return 1;
61  }
```

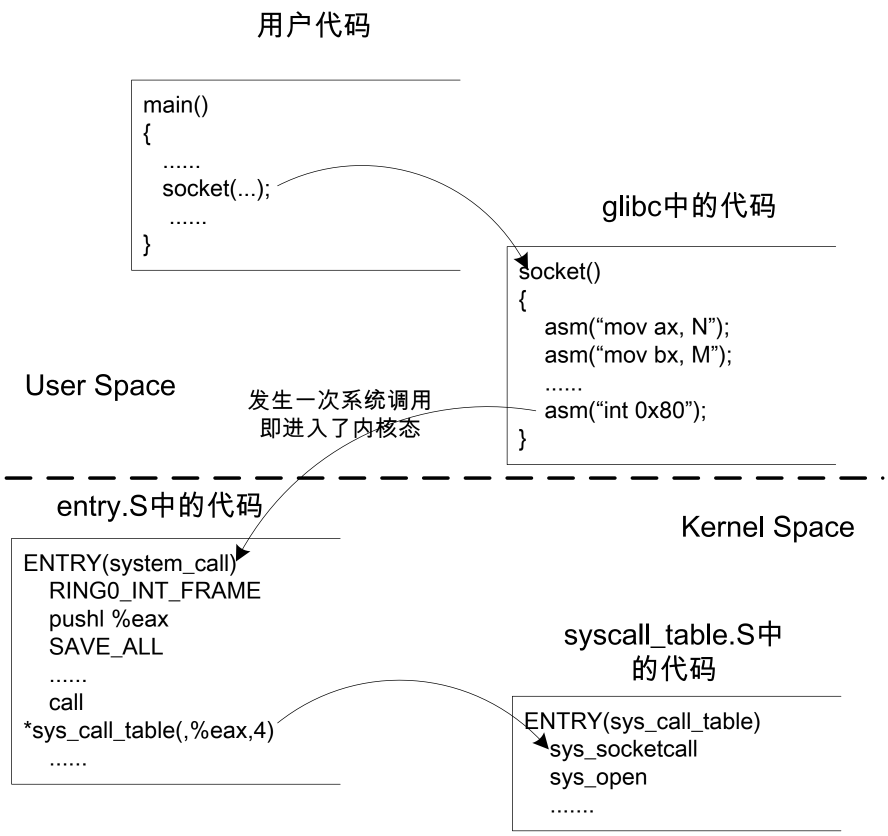
代码段 2-7do_IRQ 函数

把 action 传入了 handle_IRQ_event，然后在其中执行 action->handler，此 handle 就是每个设备驱动程序挂接的 ISR。

```
    /*
1   * 函数返回值告诉我们对此中断是否还有要处理的工作，如果有的话还得告诉软中断机制完成这些工作
    */
     asmlinkage int handle_IRQ_event(unsigned int irq,
2        struct pt_regs *regs, struct irqaction *action)
3   {
4       int status = 1; /* Force the "do bottom halves" bit */
5       int retval = 0;
6
        注意这里是一个循环，如果某个中断被指定为 SHARED，那么很有可能多个中断服务程序要处理此中断。
7       do {
8           status |= action->flags;
9           retval |= action->handler(irq, action->dev_id, regs);
10          action = action->next;
11      } while (action);
12
13      return retval;
14  }
```

代码段 2-8handle_IRQ_event 函数

上面介绍的是硬件过来的中断，而传说中的软件中断（不是软中断）是怎么工作的呢？其实很简单，下面这副图是否能满足您的求知欲呢？那些曲线箭头表示代码的执行路径。要注意的是在 socket( )函数的实现过程中，那些 mov 指令就是告诉内核要跳转的系统调用函数以及用户待传入内核的参数地址。不同的 CPU 结构上会使用不同的寄存器，我这里就不详细说明了。



图表 2-9 系统调用发生的情况

所以，软件中断的处理方式和硬件的处理路径是完全不一样的，它不必经过 do_IRQ 这个函数，而是直接跳转到内核中的代码执行 sys_socketcall。在 2.6.18 的内核中，BSD 网络接口的方式已经变成了使用 sys_socketcall 来解复用不同的系统调用，这样做的好处是减少系统调用表的大小，可以集中管理网络方面的 API：

```
1   asmlinkage long sys_socketcall(int call, unsigned long __user *args)
2   {
3       unsigned long a[6];
4       unsigned long a0,a1;
5       int err;
6       copy_from_user(a, args, nargs[call]);
7       ......
8
9       a0=a[0];
10      a1=a[1];
11
12      switch(call)
13     {
14          case SYS_SOCKET:
15              err = sys_socket(a0,a1,a[2]);
16                              break;
17          case SYS_BIND:
18              err = sys_bind(a0,(struct sockaddr __user *)a1, a[2]);
19              break;
20          case SYS_CONNECT:
21              err = sys_connect(a0, (struct sockaddr __user *)a1, a[2]);
22              break;
23          case SYS_LISTEN:
24              err = sys_listen(a0,a1);
25              break;
26          case SYS_ACCEPT:
27              err = sys_accept(a0,(struct sockaddr __user *)a1, (int __user *)a[2]);
28              break;
```

```
29          ......
30          case SYS_SEND:
31              err = sys_send(a0, (void __user *)a1, a[2], a[3]);
32              break;
33          case SYS_SENDTO:
34              err = sys_sendto(a0,(void __user *)a1, a[2], a[3],
35                       (struct sockaddr __user *)a[4], a[5]);
36              break;
37          case SYS_RECV:
38              err = sys_recv(a0, (void __user *)a1, a[2], a[3]);
39              break;
40          case SYS_RECVFROM:
41              err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],
42                         (struct sockaddr __user *)a[4], (int __user *)a[5]);
43              break;
44          ......
45          case SYS_SENDMSG:
46              err = sys_sendmsg(a0, (struct msghdr __user *) a1, a[2]);
47              break;
48          case SYS_RECVMSG:
49              err = sys_recvmsg(a0, (struct msghdr __user *) a1, a[2]);
50              break;
51          default:
52              err = -EINVAL;
53              break;
54          }
55      return err;
56  }
```

代码段 2-9 sys_socketcall 函数

目前为止，我们我们讨论的都是真正的中断，那么什么是软中断呢？请读者回顾在中断即将退出的时候会调用 irq_exit，它内部会判断是否还有中断要处理，如果已经没有了就调用 invoke_softirq，这是一个宏，它被定义成 do_softirq，此函数最终调用__do_softirq，这也就是说，实际上软中断是在处理完所有中断之后才会处理的。而且处理软中断的时候还是处于中断上下文中。不过有一些限制，详见下面对__do_softirq 代码的分析。

在目前 Linux 内核中定义了 6 种软中断，而且告诫我们不要轻易的再定义新的软中断，原话如下：

```
PLEASE, avoid to allocate new softirqs, if you need not _really_ high frequency threaded
job scheduling. For almost all the purposes tasklets are more than enough. F.e. all serial
device BHs et
  al. should be converted to tasklets, not to softirqs.
```

虽然系统中定义了 6 种软中断，但在 start_kernel 函数中调用的 softirq_init，只初始化了 2 个，分别如下：

```
1   void __init softirq_init(void)
2   {
3   open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
4   open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
5   }
```

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    SCSI_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

代码段 2-10softirq_init 函数

软中断向量 0（即 HI_SOFTIRQ）用于实现高优先级的软中断，软中断向量 3（即 TASKLET_SOFTIRQ）则用于实现诸如 tasklet 这样的一般性软中断。

Tasklet 机制是一种较为特殊的软中断。Tasklet 一词的原意是"小片任务"的意思，这里是指一小段可执行的代码，且通常以函数的形式出现。软中断向量 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 均是用 tasklet 机制来实现的。

从某种程度上讲，tasklet 机制是 Linux 内核对 BH 机制的一种扩展。在 2.4 内核引入了 softirq 机制后，原有的 BH 机制正是通过 tasklet 机制这个桥梁来纳入 softirq 机制的整体框架中的。正是由于这种历史的延伸关系，使得 tasklet 机制与一般意义上的软中断有所不同，而呈现出以下两个显著的特点：

1. 与一般的软中断不同，某一段 tasklet 代码在某个时刻只能在一个 CPU 上运行，而不像一般的软中断服务函数（即 softirq_action 结构中的 action 函数指针）那样??在同一时刻可以被多个 CPU 并发地执行。

2. 与 BH 机制不同，不同的 tasklet 代码在同一时刻可以在多个 CPU 上并发地执行，而不像 BH 机制那样必须严格地串行化执行（也即在同一时刻系统中只能有一个 CPU 执行 BH 函数）。

Bottom Half 机制在新的 softirq 机制中被保留下来，并作为 softirq 框架的一部分。其实现也似乎更为复杂些，因为它是通过 tasklet 机制这个中介桥梁来纳入 softirq 框架中的。实际上，软中断向量 HI_SOFTIRQ 是内核专用于执行 BH 函数的。

```
1   void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
2   {
3   softirq_vec[nr].data = data;
4   softirq_vec[nr].action = action;
5   }
```

代码段 2-11open_softirq 函数

我们会发现__do_softirq 内部最多只处理 10 个软中断，如果系统内部的软中断事件太多，那么就会通知 ksoftirqd 内核线程处理软中断。这样，就不会占用太多的中断上下文执行时间。

```
1   asmlinkage void __do_softirq(void)
2   {
3       struct softirq_action *h;
4       __u32 pending;
5       int max_restart = MAX_SOFTIRQ_RESTART;  //这个宏的值是 10
6       int cpu;
7
8       pending = local_softirq_pending();
9
10  restart:
11
12      local_irq_enable();
13
14      h = softirq_vec;
15
16      do {
17          if (pending & 1) {
18              h->action(h);
19          }
20          h++;
21          pending >>= 1;
22      } while (pending);
23
24  local_irq_disable();
25
26  pending = local_softirq_pending();
    如果发现还有未处理的软中断，而且没有超过 10 个，就继续处理
27  if (pending && --max_restart)
28       goto restart;
    如果已经处理了 10 个软中断，还是有软中断事件没有处理，那么就通知软中断内核守护进程（内部调用
    wake_up_process 函数）
29  if (pending)
30       wakeup_softirqd();
31
32  trace_softirq_exit();
33
34  account_system_vtime(current);
35  _local_bh_enable();
36  }
```

```
ksoftirqd()
{
    while(1)
    {
        如果没有软中断事件就重新调度系统
        if (!local_softirq_pending())
            schedule();

        while(local_softirq_pending())
        {
            又重新执行软中断处理函数
            do_softirq();
        }
    }
}
```

代码段 2-12__do_softirq 函数

k’softiqrd 内核线程属于 Linux 系统必需的部分，它在系统初始化的时候就被创建了。大家在裁减 Linux 时注意不要把这部分代码给删除了。

关于这两个软中断的技术，可以查阅相关文档，而且我们在后面关于报文接收的章节中还要提到软中断，这里就不多说了。

2.3.1.2.设备驱动挂接 ISR

设备驱动程序要处理硬件中断，必须挂接 ISR，则挂接一个 ISR 可以用这个函数：

```
1   /**
2    *  request_irq - allocate an interrupt line
3    *  @irq: 要申请的中断号
4    *  @handler: 就是传说中的 ISR
5    *  @irqflags: 中断类型标志
6    *  @devname: 该设备的名字，可以由人看懂的字符
7    *  @dev_id: 似乎是一个 ID，但是实际上在我们要讨论的网络设备中就是 net_device{}结构的一个指针
8    *
9    *  Flags:
10   *  SA_SHIRQ        中断是共享的
11   *
12   *  SA_INTERRUPT        需要禁止本地中断*
13   *
14   */
15
16  int request_irq(unsigned int irq,
17      irqreturn_t (*handler)(int, void *, struct pt_regs *),
18      unsigned long irqflags,
19      const char * devname,
20      void *dev_id)
21  {
22  int retval;
23  struct irqaction * action;
24
25  .....
26
27  action = (struct irqaction *)
28          kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
29  ......
    此 handler 就是刚才介绍的 handle_IRQ_event 函数中要处理的 handler。
30  action->handler = handler;
31  action->flags = irqflags;
32  action->mask = 0;
33  action->name = devname;
34  action->next = NULL;
35  action->dev_id = dev_id;
36
37  retval = setup_irq(irq, action);
38
39  return retval;
40  }
```

代码段 2-13request_irq 函数

要注意的是你在挂接 ISR 之前要正确的初始化你的设备，并且要保证用正确的顺序挂接中断。里面调用 setup_irq 就是把第 27 行创建的 irqaction{}挂接到对应中断的链表上，以至于 handle_IRQ_event 能根据 irq 号直接找到对应 handler。里面的实现我们就不详细说了，有兴趣的同学可以自己研究研究。

2.3.2 各种语境下的切换

某一个进程只能运行在用户方式（user mode）或内核方式（kernel mode）下。用户程序运行在用户方式下，而系统调用运行在内核方式下。2.6 调度系统从设计之初就把开发重点放在更好满足实时性和多处理机并行性上，并且基本实现了它的设计目标。

新调度系统的特性概括为如下几点：

- 继承和发扬 2.4 版调度器的特点：
  - 交互式作业优先
  - 轻载条件下调度/唤醒的高性能

  - o 公平共享
  - o 基于优先级调度
  - o 高 CPU 使用率
  - o SMP 高效亲和
  - o 实时调度和 cpu 绑定等调度手段
- 在此基础之上的新特性：
  - o O(1)调度算法，调度器开销恒定（与当前系统负载无关），实时性能更好
  - o 高可扩展性，锁粒度大幅度减小
  - o 新设计的 SMP 亲和方法
  - o 优化计算密集型的批处理作业的调度
  - o 重载条件下调度器工作更平滑
  - o 子进程先于父进程运行等其他改进

在 2.6 中，就绪队列定义为一个复杂得多的数据结构 struct runqueue，并且，尤为关键的是，每一个 CPU 都将维护一个自己的就绪队列，--这将大大减小竞争。

对于某一个特定的进程，它必定处于下面状态中的一个：

宏定义	值	含义
TASK_RUNNING	0	正在运行的进程（是系统的当前进程）或准备运行的进程（在 Running 队列中，等待被安排到系统的 CPU）。处于该状态的进程实际参与了进程调度
TASK_INTERRUPTIBLE	1	处于等待队列中的进程，待资源有效时唤醒，也可由其它进程被信号中断、唤醒后进入就绪状态
TASK_UNINTERRUPTIBLE	2	处于等待队列中的进程，直接等待硬件条件，待资源有效时唤醒，不可由其它进程通过信号中断、唤醒
TASK_STOPPED	4	进程被暂停，通过其它进程的信号才能唤醒。正在调试的进程可以在该停止状态
TASK_ZOMBIE	8	终止的进程，是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在 Task 向量表中仍有一个 task_struct 数据结构项。它不进行任何调度或状态转换，等待父进程将它彻底释放

2.3.3 内核下的同步与互斥

同步与互斥是有区别但又互相联系，在经典的操作系统教材中两个术语可以互换，为什么？因为同步是建立在互斥的基础之上的。只有实现了互斥功能，才能实现同步机制，它们之间的关系有点类似于 TCP 和 IP 的关系。在现实的操作系统实现中，这两者被严格的区分开来。同步一般用 semaphore 表示，互斥一般用 spin lock 来表示。Windows 内核源代码也是如此，linux 内核中亦然。主要 的区别是在 semaphore 机制中，当某进程进不了临界区时会进行其它进程的调度，而 spin_lock 刚执行忙等（在 smp 中是这样，但在单一 CPU 环境下则是空语句，我们会在下面呈现给大家看）。我们知道，内核中的执行路径主要有：

1 用户进程的内核态，此时有进程 context，主要是代表进程在执行系统调用 等。

2 中断或者异常或者自陷等，从概念上说，此时没有进程 context，不能进行 context switch。

3 软中断，从概念上说，此时也没有进程 context。

4 同时，相同的执行路径还可能在其他的 CPU 上运行。

这样，考虑这四个方面的因素，通过判断我们要互斥的数据会被这四个因素中的哪几个来存取，就可以决定具体使用哪种形式的锁。

2.3.3.1.中断相关的锁

local_irq_disable/local_irq_enable，表示只是对当前执行上下文的 CPU 进行开/关中断。如果在多 CPU 情形下，不保证其他 CPU 会响应中断。

2.3.3.2.spin_lock/spin_inlock

Spin_lock 采用的方式是让一个进程运行，另外的进程忙等待，由于在只有一个 cpu 的机器(UP)上微

观上只有一个进程在运行。所以在 UP 中，spin_lock 和 spin_unlock 就都是空的了。本文并不打算涉及 SMP 及多核等技术的讨论，有兴趣的读者可以自己在完成研究。在本书中，凡是遇到了 spin_lock 及其变体，基本都无视而过。

spinlock_XXX 有很多形式，有

spin_lock()/spin_unlock()，
spin_lock_irq()/spin_unlock_irq()，
spin_lock_irqsave/spin_unlock_irqrestore()
spin_lock_bh()/spin_unlock_bh()
local_bh_disable/local_bh_enable

那么，在什么情况下具体用哪个呢？这要看是在什么内核执行路径中，以及要与哪些内核执行路径相互斥。

- 如果只要和其他 CPU 互斥——spin_lock/spin_unlock，
- 如果要和 irq 及其他 CPU 互斥—— spin_lock_irq/spin_unlock_irq，
- 如果既要和 irq 及其他 CPU 互斥，又要保存 EFLAG 的状态，——spin_lock_irqsave/spin_unlock_irqrestore，
- 如果 要和 bh 及其他 CPU 互斥——spin_lock_bh/spin_unlock_bh，
- 如果不需要和其他 CPU 互斥，只要和 bh 互斥，——local_bh_disable/local_bh_enable。

本来不想讲述太多关于 spin_lock 的东西，但是由于在检查其代码的时候花费了我大量的时间，如果不记下来，以后又忘记了，多可惜！于是，我不得不花点笔墨来描述其结构。

Spin_lock 在代码中有好多定义，但其实就是_spin_lock，它在 spinlock.c 中实现，（为什么我选择“Preempt”了还是要定义成：

```
void __lockfunc _spin_lock(spinlock_t *lock)
{
    preempt_disable();
    _raw_spin_lock(lock);
}s
```

我就不知道了），_raw_spin_lock 会被定义成

define _raw_spin_lock(lock) __raw_spin_lock(&(lock)->raw_lock)，注意，那个#号和 define 中间有一个空格，会让我们有一个错觉，似乎这是一句不会被编译的代码，但是它确实是符合 C 编译要求的语句。然后我们回到 spinlock.h 的上部，发现这么几句代码：

```
#if defined(CONFIG_SMP)
# include <asm/spinlock.h>
#else
# include <linux/spinlock_up.h>
#endif
```

那么根据我本人的机器，它是单核，不是什么 SMP，而是 UP，所以我们会在 spinlock_up.h 中找到__raw_spin_lock 的定义：

define __raw_spin_lock(lock) do { (void)(lock); } while (0)

看到没有？这就是 spin_lock 在单一处理器上的实现，它是一个空语句。什么都不执行！这也就是本书中为什么无视它的原因。

2.3.3.3.down/up 信号

内核中的 semaphore 机制。它主要通过 down()和 up()两个操作实现。down() 用于获取资源，而 up()是释放资源，为什么不叫更形象的名字呢？比如 take 和 give，我也不知道 Linus 是如何想出这么奇怪的名字。一个任务通过调用 down()获取资源，而代表该资源的信号量表示“没有可用资源”的时候，进程转入等待状态，直 到占有资源的进程调用 up()释放资源后才能被唤醒。进程的等待与唤醒通过等待队列实现。up()释放一个资源实例，并且唤醒一个等待进程。若此时还有其它进程处于等待状态的，down()返回前的 wake_up()会暂时地唤醒等待进程，将 count、sleepers 再调整为(-1,1)，表示暂时无足够资源提供，然后又进入等待状态。

可以看出，semaphore 和 spin_lock 机制解决的都是两个进程的互斥问题，都是让一个进程退出临界

区后另一个进程才进入的方法，不过 sempahore 机制实行的是让进程暂时让出 CPU，进入等待队列等待的策略，而 spin_lock 实行的却是却进程在原地空转，等着另一个进程结束的策略。

2.3.3.4.RCU 读写锁

Linux2.6 还引入了新的锁机制 RCU(Read-Copy Update) 的实现机制，RCU(Read-Copy Update)，顾名思义就是读-拷贝修改，它是基于其原理命名的。对于被 RCU 保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作。

因此 RCU 实际上是一种改进的 rwlock，读者几乎没有什么同步开销，它不需要锁，不使用原子指令，因此不会导致锁竞争，内存延迟以及流水线停滞。不需要锁也使得使用更容易，因为死锁问题就不需要考虑了。写者的同步开销比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其它写者的修改操作。读者必须提供一个信号给写者以便写者能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测读者的信号，一旦所有的读者都已经发送信号告知它们都不在使用被 RCU 保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。 RCU 与 rwlock 的不同之处是：它既允许多个读者同时访问被保护的数据，又允许多个读者和多个写者同时访问被保护的数据（注意：是否可以有多个写者并行访问取决于写者之间使用的同步机制），读者没有任何同步开销，而写者的同步开销则取决于使用的写者间同步机制。但 RCU 不能替代 rwlock，因为如果写比较多时，对读者的性能提高不能弥补写者导致的损失。

2.3.4 各种异步手段

2.3.4.1.Timer（定时器函数）

作为一个有经验的开发者，一定知道所谓的定时器函数，其实都是一些回调函数而已，只不过本书中涉及到的定时器都是在内核中执行，内核中的 Timer 不是线程，它们运行在中断级，所以 timer 函数不应该做任何精细的工作。如果需要进一步处理，那么应该在 tasklet 里完成，因为 tasklet 可以被中断抢占。那么它是如何使用的呢？其实创建 Timer 的方式有好几种，出于篇幅的原因，我只举一个在协议栈经常使用的方式：

1) 定义一个 timer_list{} 结构比如名叫 atimer
2) 调用 init_timer（&atimer）
3) 指定 atimer.expires 为执行周期，atimer.function 为回调函数，timer.data 为回调函数的参数
4) add_timer(&atimer)
5) 当 atimer.expires 之后，执行回调，并且在回调函数中再执行 4)，依次重复

为什么要采用这种方式而非其它，是因为我们要保证，这种 Timer 的优先级高于其它方式创建的 Timer。

其内在机制是怎么运作的呢？答案是：软中断！其调用关系如下：

run_timer_softirq→ __run_timers

在__run_timers 函数中，先扫描用上面方法创建的 timer，然后再扫描其它的 timer。

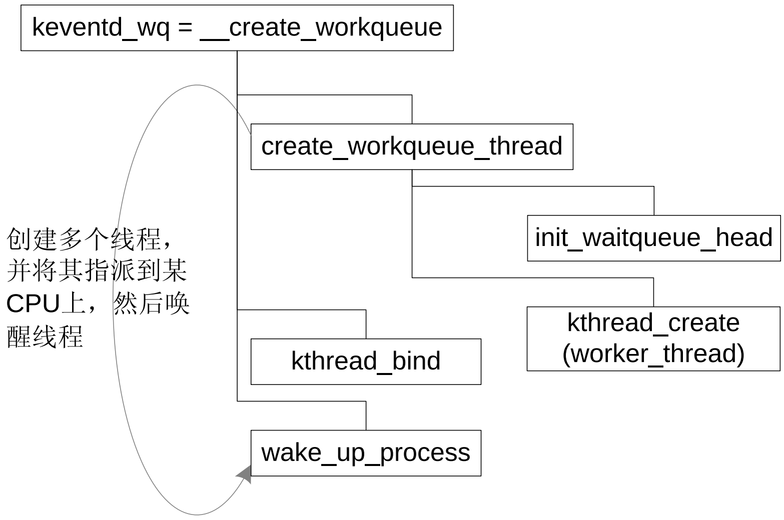
2.3.4.2.work queue（工作队列）

Work queue 是内核很早就增加的功能，类似于 Timer，指定一个回调，然后挂接到一个特殊的队列，让系统在适当的时机调用它们。它与进程调度机制紧密结合，能够用于实现内核中异步事件通知机制。其中一种使用方法如下：

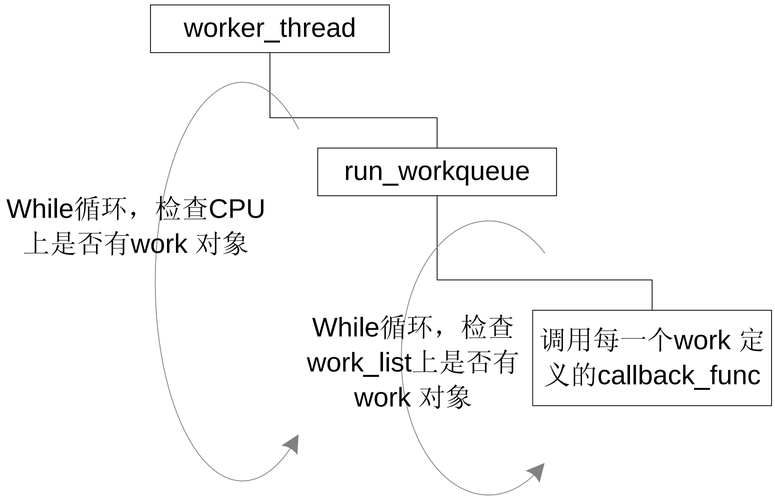
1) 定义一个 work：DECLARE_WORK(my_work, my_event_callback_func, NULL);
2) 调用 schedule_work(my_work)，那么 my_work 会在将来某个时刻被处理

Schedule_work 最终会调用 wake_up 函数去唤醒以 queue 作为等待队列头的所有等待队列对应的进程。要注意凡是调用这个函数去挂接 work struct{}时，实际上是挂到一个全局队列上：keventd_wq。读者应该会发现在 do_basic_setup 时候 Linux 曾经调用 init_workqueues 去创建一个名为 keventd_wq 的全局对象，刚才我们看到的 schedule_work 函数就是把 my_work 挂接到此对象上。在创建 keventd_wq 的时候就创建了一个内核线程。

init_workqueu 的内部过程如下：



那么创建出来的 worker_thread 的内部如下：



内核线程是被 kernel_thread [arch/i386/kernel/process]创建的,它又通过调用著名的 clone 系统调用[arch/i386/kernel/process.c] (类似 fork 系统调用的所有功能都是由它最终实现)。

下面是一些最常用的内核线程(你可以用 ps -x 命令):

PID　COMMAND
1　init
3　ksoftirqd/0
4　events/0
6　kthread
88　bdflush
90　kswapd0

读者看到第 4 号内核线程 events 就是本书要介绍的协议栈常用的 worker queue。

2.3.4.3.通知链

Linux 内核协议栈大量使用通知链这种技术，用法如下：

1） 定义一个通知链 BLOCKING_NOTIFIER_HEAD(my_event_chain)。
2） 定义一个通知块 struct notifier_block my_event_block = {
.notifier_call =my_event_process_func, };
3） 使用 blocking_notifier_chain_register(my_event_block)把这个通知块挂接到 my_event_chain。
4） 另一处代码调用 blocking_notifier_call_chain(my_event_chain)，那么就把一个事件发送到对应的通知块上，对应的处理函数就会处理事件。

内部实现是这样的 blocking_notifier_call_chain 内部调用 notifier_call_chain，这个函数内部会遍历通知链，然后以此执行通知块上的回调函数：

```
1.  int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
2.  {
3.  int ret=NOTIFY_DONE;
4.  struct notifier_block *nb = *n;
5.
6.  while(nb)
7.  {
8.      ret=nb->notifier_call(nb,val,v);
        如果执行环境处于调试中断中，那么可以退出遍历
9.      if(ret&NOTIFY_STOP_MASK)
10.     {
11.         return ret;
12.     }
13.     nb=nb->next;
14. }
15. return ret;
16. }
```

代码段 2-14　notifier_call_chain 函数

从代码上看，所谓通知链并不是真正的“通知”方式，而是直接回调方式，这就要保证不写出递归很多次的代码，以免内核栈被撑死。

2.4　虚拟文件系统

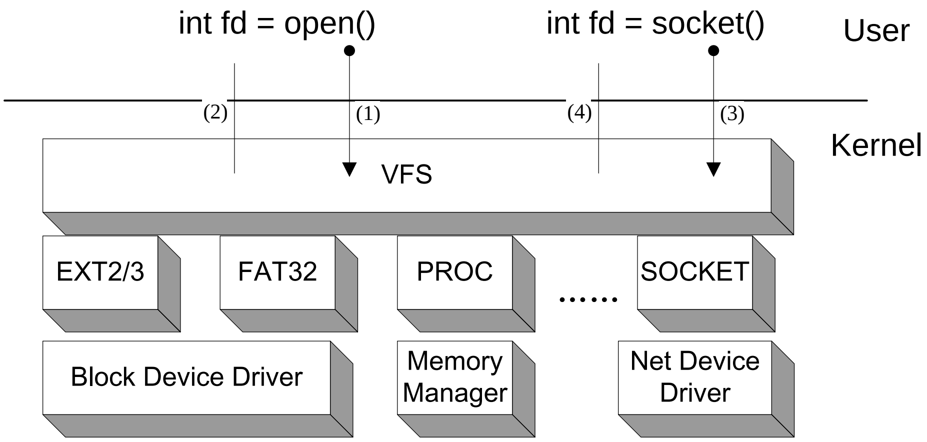
VFS（虚拟文件系统）在 Linux 及 Unix 家族中是非常重要的概念，可以说它是操作系统的骨架。每个单独的技术，比如设备驱动程序、文件、管道甚至我们要介绍的协议栈，都作为其附着物存在于内核中。不过限于篇幅，我就不一一阐述了。与任何一种 UNIX 一样，Linux 并不通过设备标识访问某个文件系统（如 DOS 那样），而是将它们“捆绑”在一个树型结构中，文件系统安装时（mount），Linux 将它挂到树的某个节点（即目录），文件系统的所有文件就是该目录下的文件或子目录，直到文件系统卸载（umount），这些文件或子目录才自然脱离。

VFS 在大多数文档中作为 Virtual File System 的缩写，我觉得还不足以点明 VFS 的作用，反而觉得用 Virtual Filesystem Switcher 来定义 VFS 可能使你更明白它的用途。是的，就是虚拟文件系统切换器。VFS 只存在与内存中，它在系统启动时被创建，系统关闭时注销。VFS 的作用就是屏蔽各类文件系统的差异，给用户、应用程序、甚至 Linux 其他管理模块提供统一的接口集合。管理 VFS 数据结构的组成部分主要包括超级块和 inode。

为什么在研究协议栈的书中要介绍 VFS 呢？因为 Linux 将网络接口也作为一个文件去操作，如果我们对文件系统不了解，那么不会很好的去解释为什么同样的发送过程，可以用 send( )，也可以用 write( )？

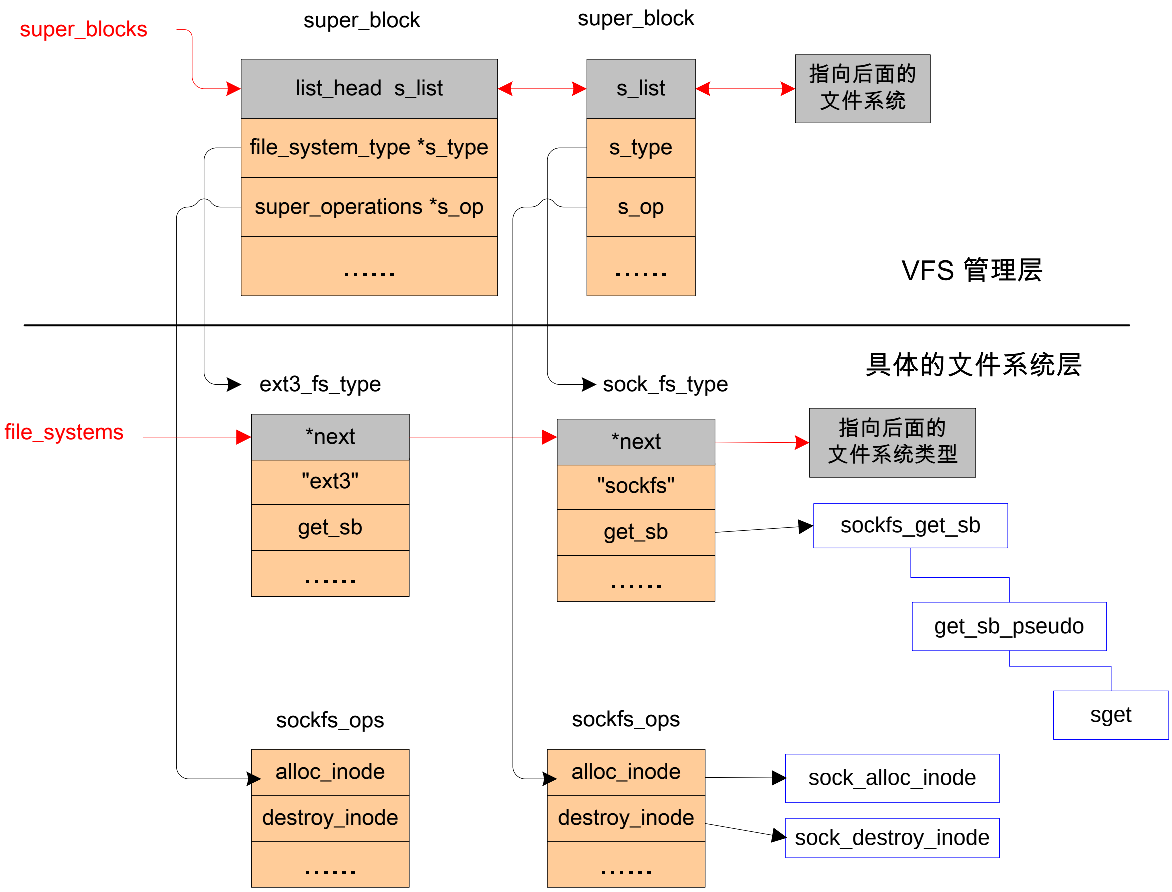
VFS 是物理文件系统与服务之间的一个接口层，它对 Linux 的每个文件系统的所有细节进行抽象，使得不同的文件系统在 Linux 核心以及系统中运行的进程看来都是相同的。严格的说，VFS 并不是一种实际的文件系统。它只存在于内存中，不存在于任何外存空间。VFS 在系统启动时建立，在系统关闭时消亡。

VFS 使 Linux 同时安装、支持许多不同类型的文件系统成为可能。VFS 拥有关于各种特殊文件系统的公共界面，当某个进程发布了一个面向文件的系统调用时，内核将调用 VFS 中对应的函数，这个函数处理一些与物理结构无关的操作，并且把它重定向为真实文件系统中相应的函数调用，后者用来处理那些与物理结构相关的操作。下图就是逻辑上对 VFS 及其下层实际文件系统的组织图，可以看到用户层只能于 VFS 打交道，而不能直接访问实际的文件系统，比如 EXT2、EXT3、PROC，换句话说，就是用户层不用也不能区别对待这些真正的文件系统，不过，SOCKET 虽然也属于 VFS 的管辖范围，但是有其特殊性，就是不能象打开大部分文件系统下的“文件”一样打开 socket，它只能被创建，而且内核中对其有特殊性处理。第 3 章我们会看到如何特殊。



图表 2-10VFS 与底层各模块关系

VFS 描述文件系统使用超级块和 inode 的方式，所谓超级块就是对所有文件系统的管理机构，每种文件系统都要把自己的信息挂到 super_blocks 这么一个全局链表上。内核中是分成 2 个步骤完成：首先每个文件系统必须通过 register_filesystem 函数将自己的 file_system_type 挂接到 file_systems 这个全局变量上，然后调用 kern_mount 函数把自己的文件相关操作函数集合表挂到 super_blocks 上。每种文件系统类型的读超级块的例程（get_sb）（ 2.4.0 的子例程名为 read_super）必须由自己实现。我们会在后面的网络文件系统初始化一节中再次研究初始化过程。



图表 2-11 super_blocks 和 file_systems 链表

文件系统由子目录和文件构成。每个子目录和文件只能由唯一的 inode 描述。inode 是 Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁。VFS inode 的内容取自物理设备上的文件系统，由文件系统指定的操作函数(i_op 属性指定)填写。VFS inode 只存在于内存中，可通过 inode

缓存访问。

Linux 即支持多种类型的文件系统，又保持极高的时空性能，究其原因，在于各种复杂缓存的作用。比如 inode 缓存。随着文件的读入和写出，这些文件的 inode 构成一条链表。一般，通过对 inode 链表的线形搜索，可以找到任一表示某设备上一个文件或子目录的 inode。从效率方面，VFS 为已分配的 inode 构造缓存和 hash 表。

Linux 下有很多种文件系统，可以通过查看文件属性推出某文件属于某个文件系统编制，使用#ls –la 命令，列出的文件信息第一栏有这样的表示方法：drwx。这是文件的 mode 字段。大部分文件系统的表面区别大致如下：

- 普通文件

Mode 字段的第一个字符用横线表示，例如,-rw-rw-rw，普通文件包含的数据内核并不感兴趣，所以不会对其文件内容改写（用户改写是另一回事）

- 目录文件

Mode 字段的第一个字符用‘d’表示，例如,drw----，其中存放着文件名和文件缩引节点之间的关联关系。

- 块设备

Mode 字段的第一个字符用‘b’表示，例如,brw----，这些文件表示一个硬件设备，通常读取这些设备的数据是通过块操作进行，比如磁盘驱动和磁带驱动文件。这些文件一般放在/dev 目录下。而且用户不能用文本查看的方式打开它们。

- 字符设备

Mode 字段的第一个字符用‘c’表示，例如,crw----，这些文件也表示一个硬件设备，通常读取这些设备的数据是通过字节流方式操作进行，比如终端输入设备驱动和串口驱动文件。这些文件一般也放在/dev 目录下。用户也不能用文本查看的方式打开它们。还有一种伪设备或设备驱动程序也是字符设备，但它们并不表示一个硬件，只是用于特殊目的，比如某个程序想和内核沟通，可以采取这样的方法。

- 链接（link）

Mode 字段的第一个字符用‘l’表示，例如,lrw----。这表示这个文件是指向另一个文件的“指针”（和内存中的指针不一样）。

- 命名管道

Mode 字段的第一个字符用‘p’表示，例如,prw----。管道也可以看作一个文件，一般用来做进程间数据通信，与设备文件比较相似，它的工作方式是 FIFO。

- 套接字

终于到了和我们比较相关连的文件类型了，其 Mode 字段的第一个字符用‘s’表示，例如,srw----。不过，在这里套接字文件是用来给不同机器间不同进程间消息交互的，似乎有点与我们的协议栈没太多关系。

还有一些特殊的文件，不代表文件也不代表设备，只是用来提供内核数据和地址空间的访问，如/proc，这个反而是我们要经常关注的文件系统。

> 以上章节是理解内核的基础，限于篇幅，不能再讲得太多，否则喧宾夺主。这里要强调得是，以上每一小节，都可以写出很多页甚至单独成书。希望有兴趣的读者自行到 Internet 上去搜索相关内容。以下和网络具体实现的内容关系则更加密切。

2.5 网络协议栈各部分初始化

协议栈本色的初始化这部分在 2.6 早期和后期是不太一样的，早期的是通过函数直接调用的方式，后期更加依赖于使用 init.h 中定义的初始化宏来进行，即放入特定的代码段去执行。所以再次强调关于内核镜像研究的重要性。我们先给出初始化大致的顺序，大家记住这个顺序就对初始化有关全局的了解：

① core_initcall：sock_init

② fs_initcall：inet_init

③ subsys_initcall：net_dev_init

④ device_initcall:设备驱动初始化

上面那句话揭示了初始化的顺序，读者可以看到源代码中sock_init，inet_init，设备驱动初始化是分别被core_initcall、fs_initcall、subsys_initcall 、device_initcall函数修饰的，根据前文对初始化section的描述，这四个函数放在不同的section，而且执行顺序是从①到④的。所以我们对初始化的描述也是按照这个顺序进行。切记，分析代码万不可颠三倒四。

2.5.1 网络基础系统初始化

先看看第一个步骤完成什么功能：使用 core_initcall 初始化宏修饰 sock_init 函数，这个宏指定了 sock_init 函数放在级别为 1 的代码段中，也就是说它的执行是最先进行的一部分（在早期的代码中是先从 start_kernel 函数开始，调用 kernel_thread 启动了 init 进程（在同一个文件里），再调用 sock_init）。此函数只是分配一些内存空间，以及创建了一个 sock_fs_type 的文件系统。在 do_basic_setup 中调用 sock_init 先于 Internet 协议注册被调用，因为基本的 socket 初始化必须在每一个 TCP/IP 成员协议能注册到 socket 层之前完成。

```
1.      void __init sock_init(void)
2.      {
3.      int i;
4.
5.      // 在此只是将此数组初始化为 NULL，真正的初始化要等到 inet_init 进行
6.      for (i = 0; i < NPROTO; i++)
7.          net_families[i] = NULL;
8.
        初始化 sock SLAB cache.（此函数内部没做什么事，只是对若干 sysctl_mem_max 进行赋值，估计这行注释是对老代码的解释，Linuxer 在 2.6.18 还没来得及删掉。）
9.       sk_init();
10.
        此函数是为了 socket buffer 设置 slab cache。
11.     skb_init();
12.
13.     /*
14.      *  Initialize the protocols module.
15.      */


         然后为 socket 建立伪文件系统，第一步是设置 socket inode cache。
16.     init_inodecache();
17.     register_filesystem(&sock_fs_type);
18.     sock_mnt = kern_mount(&sock_fs_type);
19.     // 当执行 do_initcalls 时才真正初始化那些特定协议协议，比如执行 inet_init
20.     ……
21.     }
```

```
static struct file_system_type sock_fs_type
= {
        .name =             "sockfs",
        .get_sb =    sockfs_get_sb,
        .kill_sb =   kill_anon_super,
};
```

代码段 2-15 sock_init 函数

所谓基础设施，就是协议栈要运作所需的基本环境，当我们把协议栈看作一个小模块时，它必须与系统中其他部分打交道。比如内存管理（在 skb_init 中完成），和上下层之间的联系（在创建 socket 文件系统的过程中完成）。本小节只是引入了这两部份的概念，后面的章节会一一介绍。

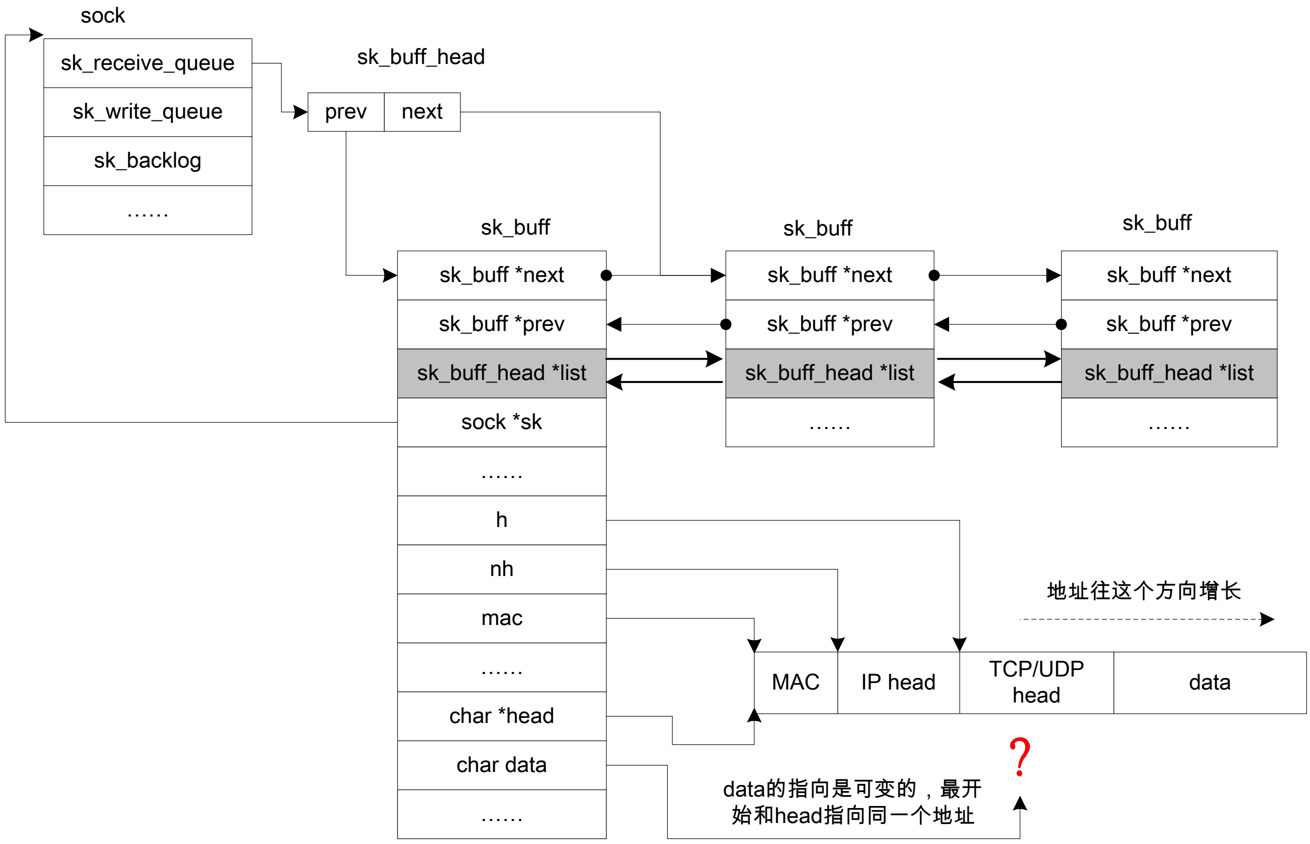
2.5.2 网络内存管理

关于内核中网络部分的内存管理，凡涉及到内核的 Linux 参考书都会提及，在本书中不会对它大书特书，只是给出一些基本的概念，读者只要能基本理解网络内部内存的管理思路即可。

2.5.2.1. sk_buff 结构

数据包在应用层称为 data，在 TCP 层称为 segment，在 IP 层称为 packet，在数据链路层称为 frame。Linux 内核中 sk_buff{}结构来存放数据，在不同的层次会使用上面几种术语来称呼，但这没有什么太大区别。sk_buff{ }在 INET Socket 和它以下的层次中用来存放网络接收到或需要发送的数据，因此它需要设计得要有足够的扩展性，从而可以支持不同层次、相同层次不同类型的网络协议。另外，还必须高效、紧凑。

网络内存管理的结构：



图表 2-12 sock 和 sk_buff 的关系

从上图中看到一个问号，表示 data 这个指针指向的位置是可变的，它有可能随着报文所处的层次而变动。当接收报文时，从网卡驱动开始，通过协议栈层层往上传送数据报，通过增加 skb→data 的值，来逐步剥离协议首部；而要发送报文时，各协议创建 sk_buff{}，在经过各下层协议时，通过减少 skb→data 的值来增加协议首部。不仅有这种手段，我们还可以通过 h,nh,mac 这三个联合指针，我们可以访问到这些协议首部，从而利用其提供的有效信息。

先看到的是 sk_buff_head{}结构，它是每个数据流的"头"，凡是能保存数据的地方，都用这个结构来指向真正的数据。

```
struct sk_buff_head {
    /* These two members must be first. */
   /* 这两个成员必须放在前面。原因在于对 sk_buff_head 进行操作的时候，可以用 sk_buff
      结构做类型的强制转换来完成，反过来一样 */
    struct sk_buff  *next;
    struct sk_buff  *prev;

    __u32       qlen; /*  该 sk_buff_head 结构引导的一个链表的节点的个数 */
};
```

代码段 2-16 sk_buff_head 结构

下面这个结构才是真正指向数据区的"指针集合"。其中 head 成员指向真正的数据区。

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff      *next;
    struct sk_buff      *prev;

    struct sk_buff_head *list;
    struct sock     *sk;
    struct timeval      stamp;
```

```
    struct net_device   *dev;
    struct net_device   *real_dev;
   下面是关于第 4 层/传输层首格式,用 h 表示这个联合的名字,(其实我觉得可以叫 th,即 transport header)
    union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
        struct iphdr    *ipiph;
        struct ipv6hdr  *ipv6h;
        unsigned char   *raw;
    } h;
    下面是关于第 3 层/网络层首格式，就是传说中的 IP 头，所以联合的名字叫 nh,即 network header
    union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
    } nh;
    下面就是 MAC 层的头
    union {
        struct ethhdr   *ethernet;
        unsigned char   *raw;
    } mac;
    以上这种安排也体现了报文各层头部的逻辑关系
    这个是路由 cache 的指针，后面的章节会着重介绍
    struct  dst_entry   *dst;
    这个是 xfrm 相关的成员，不必关心
    struct  sec_path    *sp;

    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a skb_clone()
     * first. This is owned by whoever has the skb queued ATM.
     */
    char            cb[48];

    unsigned int        len,
                data_len,
                mac_len,
                csum;
    unsigned char       local_df,
                cloned, /* 指示此 sk_buff{}是否被“克隆”过。*/
    当接收一个报文时，创建一个 sk_buff{}，然后根据地址类型指定该 skb 实际属于哪一种的报文类型，然后上
    层协议栈采取相应的处理方式处理该 skb ， 见下面的表
                pkt_type,
                ip_summed;
    __u32           priority;
    可以用 eth_type_trans 函数获取 protocol 的值，如果以太网头大于 1536，那么就返回以太网的 h_proto
    值，下面时常用值
    unsigned short      protocol,
```

宏	值	说明
ETH_P_802_2	4	真正的 802.2 LLC，当报文长度小于 1536 时
ETH_P_LOOP	0x0060	以太网环回报文
ETH_P_IP	0x0800	IP 报文
ETH_P_ARP	0x0806	ARP 报文
BOND_ETH_P_LACPDU	0x8809	LACP 协议报文
ETH_P_8021Q	0x8100	VLAN 报文
ETH_P_MPLS_UC	0x8847	MPLS 单播报文

```
    void            (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_NET_SCHED
      __u32         tc_index;           /* traffic control index */
```

```
#endif

    下面这些成员必须放在最后，在 alloc_skb() 函数中（创建 sk_buff）为了提高性能，只将上面各部分全部
清 0，而下面的部分可以后面指定。
    unsigned int        truesize;
    atomic_t        users;/* 每引用或“克隆”一次 sk_buff{ }结构的时候，都要自加 1 */
    unsigned char       *head,
                        *data,
                        *tail,
                        *end;
};
```

代码段 2-17 sk_buff 结构

len 是指数据包全部数据的长度，包括 data 指向的数据和 end 后面的分片的数据的总长，而 data_len 只包括分片的数据的长度。而 truesize 的最终值是 len+sizeof(struct sk_buff)。

如果一个 sk_buff{}是“克隆”得来的，那么它的 clone 成员和源 socket{}的 clone 成员都是 1。通过 skb_clone()函数完成一次“克隆”操作，“克隆”过后的 sk_buff{}不存在于链表中，也不和某一个特定的 INET socket 相关联，通过检查成员 users 可以知道这个 sk_buff 结构是否被“克隆”过。“克隆”数据包的好处在于可以提高数据使用的效率。

sk_buff.pkt_type 是指该数据包的类型，定义如下：

表格 2-1

宏	值	说明
PACKET_HOST	0	该报文的目的地是本机
PACKET_BROADCAST	1	广播数据包，该报文的目的地是所有主机
PACKET_MULTICAST	2	组播数据包
PACKET_OTHERHOST	3	到其他主机的数据包，在 VLAN 接口接收数据时有重要的作用
PACKET_OUTGOING	4	它不是“发送到外部主机的报文”，而是指接收的类型，这种类型用在 AF_PACKET 的套接字上，这是 Linux 的扩展
PACKET_LOOPBACK	5	MC/BRD 的 loopback 帧（用户层不可见）

为了使用套接字缓冲区，内核创建了两个后备高速缓存(lookaside cache)，它们分别是 skbuff_head_cache 和 skbuff_fclone_cache，协议栈中所使用到的所有的 sk_buff 结构都是从这两个后备高速缓存中分配出来的。两者的区别在于 skbuff_head_cache 在创建时指定的单位内存区域的大小是 sizeof(struct sk_buff)，可以容纳任意数目的 struct sk_buff，而 skbuff_fclone_cache 在创建时指定的单位内存区域大小是 2*sizeof(struct sk_buff)+sizeof(atomic_t)，它的最小区域单位是一对 strcut sk_buff 和一个引用计数，这一对 sk_buff 是克隆的，即它们指向同一个数据缓冲区，引用计数值是 0,1 或 2，表示这一对中有几个 sk_buff 已被使用。

创建一个套接字缓冲区，最常用的操作是 alloc_skb，它在 skbuff_head_cache 中创建一个 struct sk_buff，如果要在 skbuff_fclone_cache 中创建，可以调用__alloc_skb，通过特定参数进行。

struct sk_buff 的成员 head 指向一个已分配的空间的头部，该空间用于承载网络数据，end 指向该空间的尾部，这两个成员指针从空间创建之后，就不能被修改。data 指向分配空间中数据的头部，tail 指向数据的尾部，这两个值随着网络数据在各层之间的传递、修改，会被不断改动。所以，这四个指针指向共同的一块内存区域的不同位置，该内存区域由__alloc_skb 在创建缓冲区时创建。

那指向的这块内存区域有多大呢？一般由外部根据需要传入。外部设定这个大小时，会根据实际数据量加上各层协议的首部，再加 15(为了处理对齐）传入，在__alloc_skb 中根据各平台不同进行长度向上对齐。但是，我们另外还要加上一个存放结构体 struct skb_shared_info 的空间，也就是说 end 并不真正指向内存区域的尾部，在 end 后面还有一个结构体 struct skb_shared_info，下面是其定义：

```
struct skb_shared_info{
        atomic_t        dataref;    //引用计数。
        unsigned short  nr_frags;   //数据片段的数量。
        unsigned short  tso_size;
```

```
        unsigned short    tso_segs;
        unsigned short    ufo_size;
        unsigned int      ip6_frag_id;
        struct sk_buff    *frag_list;         //数据片段的链表。
        skb_frag_t    frags[MAX_SKB_FRAGS];      //每一个数据片段的长度。
    };
```

这个结构体存放分隔存储的数据片段，将数据分解为多个数据片段是为了使用分散/聚集 I/O。

如果是在 skbuff_fclone_cache 中创建，则创建一个 struct sk_buff 后，还要把紧邻它的一个 struct sk_buff 的 fclone 成员置标志 SKB_FCLONE_UNAVAILABLE，表示该缓冲区还没有被创建出来，同时置自己的 fclone 为 SKB_FCLONE_ORIG，表示自己可以被克隆。最后置引用计数为 1。

最后，truesize 表示缓存区的整体长度，置为 sizeof(struct sk_buff)+传入的长度，不包括结构 struct skb_shared_info 的长度。

2.5.2.2. 内存管理函数

在 sk_buff{}中的 4 个指针 data、head、tail、end 初始化的时候，data、head、tail 都是指向申请到的数据区的头部，end 指向数据区的尾部。在以后的操作中，一般都是通过 data 和 tail 来获得在 sk_buff 中可用的数据区的开始和结尾。而 head 和 end 就表示 sk_buff 中存在的数据包最大可扩展的空间范围。

表格 2-2

alloc_skb	申请一个 sk_buff{}结构，并且其中申请了真正的数据区。
kfree_skbmem	释放 sk_buff{}数据区，还要根据 fclone 是否清除 sk_buff{}本身
kfree_skb	封装了 kfree_skbmem，也能释放 skb
dev_alloc_skb	在真正要发送数据的时候，要针对底层的协议申请出一个 sk_buff{}空间来存放需要发送的数据包。这个函数内部调用 alloc_skb(length+16,…)函数，在这里，除了 length 的长度空间之外，还要多申请 16 个字节的长度。这是为了存放以太网上硬件头而预留的空间。RFC 规定以太网硬件长度是 14 个字节，但为了让硬件头后面的 IP 头和 32 位地址对齐，就申请了 16 个字节的空间给硬件头使用，空出两个字节。
skb_put	将数据添加到现有数据尾部，tail 指针往右移，len 要增加移动的数量
skb_push	把 data 指针往左移，增加报文头，只是把 data 减去 sizeof(struct 报文头)，同时 len 加上这个值，这样，在逻辑上,skb 包含报文头了，通过 skb->h 或 nh 或 mac 还能找到它
skb_headroom	得到该 sk_buff 数据区头部的空闲区间大小
skb_tailroom	得到该 sk_buff 数据区尾部的空闲区间大小
skb_reserve	空出一部分空间在数据区的头部
skb_pull	把 data 指针往右移，剥掉报文头，只是把 data 加上 sizeof(struct 报文头)，同时 len 减去这个值，这样，在逻辑上,skb 已经不包含报文头了，但通过 skb->h 或 nh 或 mac 还能找到它
skb_trim	把 tail 指针往左移，剥掉数据区的尾部数据，len 减去移动的数量

下面介绍关于 sk_buff 链表的操作函数：

skb_insert	在链表中插入一个 sk_buff 结构，不涉及 sk_buff_head 结构
skb_appned	在链表中指定一个 sk_buff{}后插入一个 sk_buff，也不涉及 sk_buff_head 结构
skb_queue_head	在链表头增加一个 sk_buff 节点
skb_queue_tail	在链表尾增加一个 sk_buff 节点
skb_unlink	从链表中删除一个 sk_buff 节点
skb_dequeue	从链表头取出一个 sk_buff 节点，并且删除掉此节点
skb_dequeue_tail	在链表尾取出一个 sk_buff 节点，并且从链表中删除

本书只分析一下 skb 是如何分配的。一般来讲，一个套接字缓冲区总是属于一个套接字，所以，除

了调用 alloc_skb 函数创建一个套接字缓冲区，套接字本身还要对 sk_buff 进行一些操作，以及设置自身的一些成员值。下面我们来分析这个过程。alloc_skb 调用了__alloc_skb 函数，记住，其传入的最后一个参数 fclone 总是 0。不过总有例外，alloc_skb_fclone 函数传入的是 1，那么预示着内存的分配是从 clone 区分配。

```
1.  /**
2.   *  __alloc_skb -   申请一块网络缓冲区
3.   *  @size: 要申请的大小
4.   *  @gfp_mask: 申请码
5.   *  @fclone: 是否申请一个可被克隆的 cache。如果是可被克隆的，那么还将申请一个待克隆的 skb
6.   *
7.   *  Allocate a new &sk_buff. The returned buffer has no headroom and a
8.   *  tail room of size bytes. The object has a reference count of one.
9.   *  The return is the buffer. On a failure the return is %NULL.
    当在中断里申请缓冲区时必须传入 GFP_ATOMIC 作为 gfp_mask 的参数，这意味着不允许等待并且使用紧急 pool
    中的缓冲区
10.  */
11. struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
12.             int fclone)
13. {
14. kmem_cache_t *cache;
15. struct skb_shared_info *shinfo;
16. struct sk_buff *skb;
17. u8 *data;
    由于 fclone 总是 0，所以肯定是在后者中分配空间
18. cache = fclone ? skbuff_fclone_cache : skbuff_head_cache;
19.
20. /* 分配 sk_buff 的空间，注意，只是"sk_buff" */
21. skb = kmem_cache_alloc(cache, gfp_mask & ~__GFP_DMA);
22.
23. /* 这里才是分配上图中 data 的数据区. Size must match skb_add_mtu(). */
24. size = SKB_DATA_ALIGN(size);
25. data = ____kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
26.
27. memset(skb, 0, offsetof(struct sk_buff, truesize));
28. skb->truesize = size + sizeof(struct sk_buff);
29. atomic_set(&skb->users, 1);
30. skb->head = data;
31. skb->data = data;
32. skb->tail = data;
33. skb->end  = data + size;
34. /* make sure we initialize shinfo sequentially */
35. shinfo = skb_shinfo(skb);
36. atomic_set(&shinfo->dataref, 1);
37. shinfo->nr_frags  = 0;
38. shinfo->gso_size = 0;
39. shinfo->gso_segs = 0;
40. shinfo->gso_type = 0;
41. shinfo->ip6_frag_id = 0;
42. shinfo->frag_list = NULL;
43.
44. if (fclone) {
        定义一个指针指向内存中紧邻它的下一个节点
45.     struct sk_buff *child = skb + 1;
46.     atomic_t *fclone_ref = (atomic_t *) (child + 1);
        这个标志的值是 1，而缺省的是 0，即 SKB_FCLONE_UNAVAILABLE，说明正常情况下 skb 不是 clone 的，
        这会导致在 kfree_skbmem 的行为完全不一样。另一个值 SKB_FCLONE_CLONE 是 2
47.     skb->fclone = SKB_FCLONE_ORIG;
48.     atomic_set(fclone_ref, 1);
        置下一个单元的标志
49.     child->fclone = SKB_FCLONE_UNAVAILABLE;
50. }
51. out:
52. return skb;
53. nodata:
54. kmem_cache_free(cache, skb);
```

Generic Segmentation Offload （GSO）：

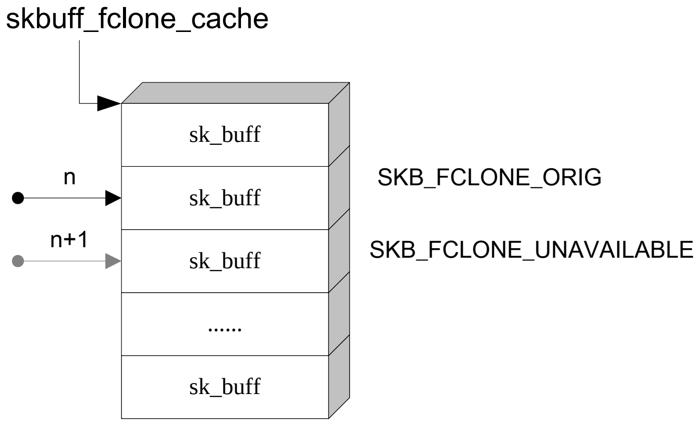
协议栈的效率提高一个策略： 尽可能晚的推迟分段（segmentation), 最理想的是在网卡驱动里分段，在网卡驱动里把大包（super-packet)拆开，组成 SG list，或在一块预先分配好的内存中重组各段，然后交给网卡。

```
55. skb = NULL;
56. goto out;
57. }
```

代码段 2-18 __alloc_skb 函数

注意，sk_buff{}来自 skbuff_head_cache 或 skbuff_fclone_cache，data 区来自普通的内核区。

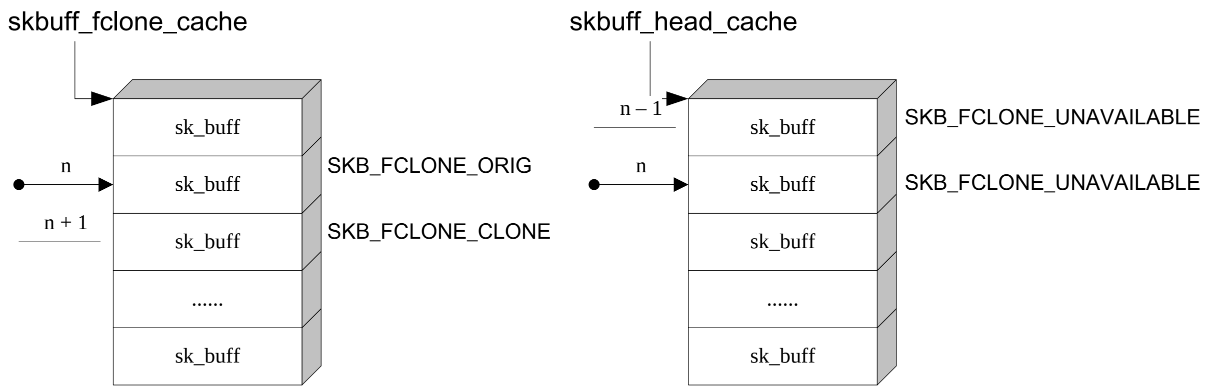
下面解释一下 alloc_skb_fclone 的操作。此函数申请一个 skb（sk_buff+data），但是这个 skb 申明自己是可以被克隆的。那么内存中看到的是这样：



图表 2-13 skbuff_fclone_cache 中的内存操作

这便是原本 sk_buff 的操作结果。用户真正需要的 skb 被设置为克隆原本的标志，而且在这个 sk_buff 的后面一个区域被设置为 SKB_FCLONE_UNAVAILABLE（0）。当用户需要 clone 一个 skb 时，就调用 skb_clone，把克隆原本的 sk_buff 后面的区域设置为 CLONE 副本标志。

但是当某 skb_buff 不可以被克隆时，那么就从 skbuff_head_cache 中申请内存，比如下图的 n－1 块。而且——读者请注意——如果某 skb_buff 确实是可以被克隆，但在它之前已经有一块区域已经不是 SKB_FCLONE_UNAVAILABLE 了，也许是 SKB_FCLONE_ORIG，也许是 SKB_FCLONE_CLONE，那么还是得从 skbuff_head_cache 中分配内存。



图表 2-14 不同 skb cache 中的内存操作

```
1.  /*
2.   *  Free an skbuff by memory without cleaning the state.
3.   */
4.  void kfree_skbmem(struct sk_buff *skb)
5.  {
6.  struct sk_buff *other;
7.  atomic_t *fclone_ref;
8.
9.  skb_release_data(skb);
10. switch (skb->fclone) {
11. case SKB_FCLONE_UNAVAILABLE:
```

```
12.     kmem_cache_free(skbuff_head_cache, skb);
13.     break;
14.
15. case SKB_FCLONE_ORIG:
16.     fclone_ref = (atomic_t *) (skb + 2);
        如果引用值大于 0，说明还有副本存在，那么什么都不做，返回。如果没有副本了，就释放原本内存
17.     if (atomic_dec_and_test(fclone_ref))
18.         kmem_cache_free(skbuff_fclone_cache, skb);
19.     break;
20.
21. case SKB_FCLONE_CLONE:
22.     fclone_ref = (atomic_t *) (skb + 1);
23.     other = skb - 1;
        把自己的 fclone 标志简单地设置为 0 即可，而不用释放，因为本来也没有“申请”内存
24.     /* The clone portion is available for fast-cloning again.*/
25.     skb->fclone = SKB_FCLONE_UNAVAILABLE;
        如果所有副本和原本都不存在了，那么就删除原本。这隐含说明了原本曾经尝试过删除自己（但未能完全成功，
        请参考前几行代码）。
26.     if (atomic_dec_and_test(fclone_ref))
27.         kmem_cache_free(skbuff_fclone_cache, other);
28.     break;
29.   };
30. }
```

代码段 2-19 kfree_skbmem 函数

好，到此可以得出 skb_clone 和 skb_copy 的区别：前者基本在 skbuff_fclone_cache 中分配内存，除非一定要对一个不是可以被克隆的对象进行克隆，那么才会在 skbuff_head_cache 中分配内存，而且只是 sk_buff{}结构的复制，没有涉及到真正数据区（data）的复制；后者必定在 skbuff_head_cache 中进行，不仅复制 sk_buff{}，而且复制了数据区。

以上是单纯得 skb 操作，每一个上层协议都实现了自己对 skb 得特殊处理。比如 TCP，如果检查到待发送数据报没有传输层协议头（不是传输层的 tcp 或 udp 数据报），套接字创建缓冲区的函数是 sock_alloc_send_skb，对于非传输层协议包，不使用分散/聚集 IO，所以，置 data_len 为 0。

上层协议利用这些内部函数，在自己本层内实现自己的内存申请和释放函数。比如说 TCP，它的相应函数分别是 tcp_alloc_pskb 和 tcp_free_skb 函数：

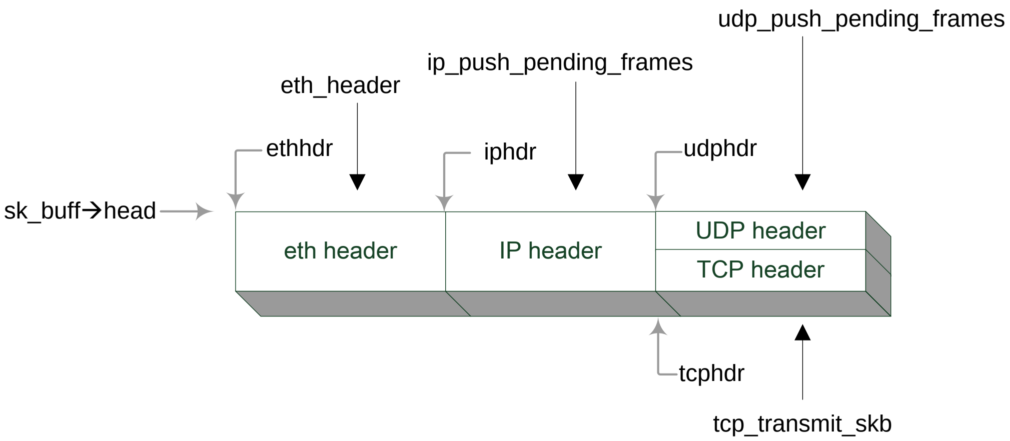
```
1.  static inline struct sk_buff *tcp_alloc_pskb(struct sock *sk, int size, int mem, int
    gfp)
2.  {
3.  struct sk_buff *skb = alloc_skb(size+MAX_TCP_HEADER, gfp);
    不仅分配分配 skb，还要把属于当前 sock 的之前分配的内存大小相加，因为 TCP 的报文很有可能被分段
4.  if (skb) {
5.      skb->truesize += mem;
6.      if (sk->sk_forward_alloc >= (int)skb->truesize ||
7.          tcp_mem_schedule(sk, skb->truesize, 0)) {
8.          skb_reserve(skb, MAX_TCP_HEADER);
9.          return skb;
10.     }
11.     __kfree_skb(skb);
12. }
13. ......
14. return NULL;
15. }
```

代码段 2-20 tcp_alloc_pskb 函数

```
1.  static inline void tcp_free_skb(struct sock *sk, struct sk_buff *skb)
2.  {
3.      tcp_sk(sk)->queue_shrunk = 1;
        仅仅释放当前 skb，不一定 sock 的报文已经被清除
4.      sk->sk_wmem_queued -= skb->truesize;
5.      sk->sk_forward_alloc += skb->truesize;
6.      __kfree_skb(skb);
7.  }
```

代码段 2-21tcp_free_skb 函数

在这里先给出对 skb 的报头基本操作，Linux 用几种数据结构分别代表不同层次的报文头：ethhdr{}、iphdr{}、udphdr{}、tcphdr{}。



图表 2-15 各协议层函数对网络报文头的理解

上图即是协议栈中各层次的代表函数处理各部分的示意图。不过不管如何转，sk_buff→head 总是指向数据的最开始。我们介绍内存管理也就到此为止了，本书余下部分不想再这个方面纠缠了，因为我本来就不想让我自己累着。

2.5.3 网络文件系统初始化

在 linux 系统中，socket 属于文件系统的一部分，网络通信可以被看作对文件的读取。这种特殊的文件系统叫 sockfs。上一节中提到在 sock_init 函数中先调用 init_inodecache，为创建 socket 文件系统做好内存准备。不过要注意的是在 Linux 内核中存在 init_inodecache 多个定义，但都是静态型，即只能由该.c 文件中的函数调用，在 socket.c 中，就定义了这么一个函数，所以 sock_init 调用就是下面这个，代码如下：

```
1.  static int init_inodecache(void)
2.  {
3.      sock_inode_cachep = kmem_cache_create("sock_inode_cache",
4.                  sizeof(struct socket_alloc),
5.                  0, SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT,
6.                  init_once, NULL);
7.      ......//错误处理
8.      return 0;
9.  }
```

代码段 2-22 init_inodecache 函数

为文件系统准备 inode 缓存部分了，下面进入初始化文件系统。

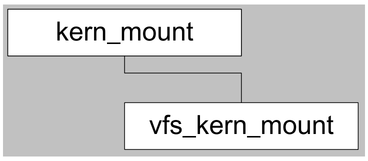
首先是调用 register_filesystem(&sock_fs_type);把文件系统类型注册到 file_systems 链表上，然后调用 kern_mount(&sock_fs_type);把该文件系统注册到 super_blocks 上。

在系统初始化的时候要通过 kern_mount 安装此文件系统。所谓创建一个套接字就是在 sockfs 文件系统中创建一个特殊文件。super_block 里面有一个字段 s_op 是用来指向某文件系统内部的支持函数，这个字段的类型为 super_operation。这个结构定义了 12 个函数指针，这些指针是要让 VFS 来调用的。因此这是 VFS 和文件系统之间的一个接口，经由这层接口，超级块可以控制文件系统下的文件或目录。有趣的

是，在 2.4.0 内核中，这个结构只有 10 个函数指针，新增的 2 个偏偏被 sockfs_ops 使用，而其他的大部分函数指针都没有被用到。要注意的是这个结构不是用户层操作 socket 内部的接口函数集合。有的读者可能有编写设备驱动程序的经验，知道如果要实现某型设备的驱动基本上要提供一个基于 file_operations{}结构的全局变量，用户层的 open、read、write 等操作都会映射到这个结构里的成员函数。但是在这里，sockfs_ops 不是提供这样的功能的，它是专门给 super_blocks 用来创建 inode 的。对于 file_operations{}，我们会在介绍 socket（）系统调用的时候作出详细说明。

```
static struct super_operations sockfs_ops = {
 .alloc_inode =  sock_alloc_inode,
 .destroy_inode =sock_destroy_inode,
 .statfs =   simple_statfs,
};
```

那么这个数据结构是如何挂到 super_blocks 上的呢？现在看看 sock_mnt = kern_mount(&sock_fs_type);内部发生了什么：



图表 2-16 kern_mount 函数调用树

看看 do_kern_mount 内部代码，这个函数在不同内核版本下不太一样，但基本操作一样：

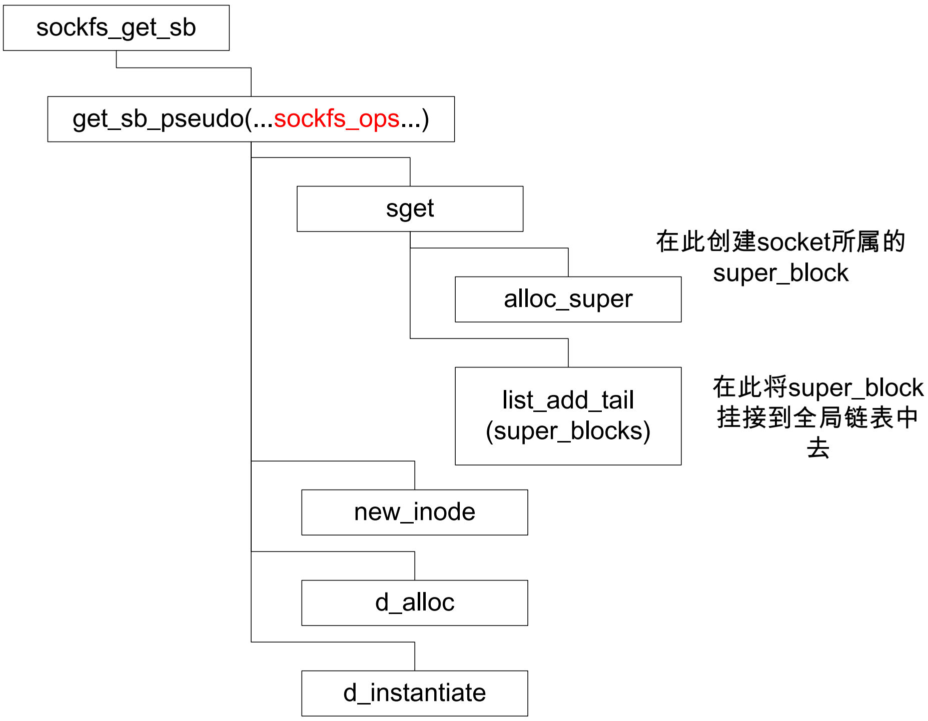
```
10. struct vfsmount *
11. do_kern_mount(const char *fstype, int flags, const char *name, void *data)
12. {
13.     struct file_system_type *type = get_fs_type(fstype);
14.     struct super_block *sb = ERR_PTR(-ENOMEM);
15.     struct vfsmount *mnt;
16.     int error;
17. 
18.     ……//出错处理
19. 
20.     mnt = alloc_vfsmnt(name);
21.     ……//出错处理
22. 
23.     if (data) {
24.         secdata = alloc_secdata();
25.         ……//出错处理
26. 
27.         error = security_sb_copy_data(type, data, secdata);
28.         ……//出错处理
29.     }
30. 
31.     sb = type->get_sb(type, flags, name, data);
32.     ……
33. 
34.     mnt->mnt_sb = sb;
35.     mnt->mnt_root = dget(sb->s_root);
36.     mnt->mnt_mountpoint = sb->s_root;
37.     mnt->mnt_parent = mnt;
38.     up_write(&sb->s_umount);
39.     put_filesystem(type);
40.     return mnt;
41. 
42.     ……//出错处理
43. }
```

```
static int sockfs_get_sb(struct file_system_type *fs_type, int flags, const char *dev_name, void *data, struct vfsmount *mnt)
```

代码段 2-23 do_kern_mount 函数

type->get_sb 实际上就是 sockfs_get_sb,，我们在前面介绍虚拟文件系统中已经看到它的身影了。现在研究它的实现，它就是把 sockfs_ops 所属的 super_block 结构挂接到全局链表 super_blocks 中。通过这

么一个操作，形成了前面的图中显示的三元组：<super_block，sock_fs_type，sockfs_ops>。这样，协议栈与用户层的连接关系基本确定。



图表 2-17 sockfs_get_sb 函数调用树

new_inode 先创建一个 inode，然后将它设置为根节点。再创建一个 dentry 结构，与之对应起来，这些内容可以参考其它关于 VFS 实现细节的资料。不管怎样，Linux 的 socket 文件系统算是建立起来了。但是我们还是不能完全不再碰见它，在下一章我们还会温习之。

2.5.4 网络协议初始化

初始化第二个大步骤就是使用 fs_initcall 初始化宏修饰 inet_init 函数，它初始化和协议本身相关的东西。第二步才真正涉及到“栈”的概念。到此我们才开始真正的网络协议的初始化。在这之前必须知道一些概念——地址族和套接字类型。大家都知道所谓的套接字都有地址族，实际就是套接字接口的种类，每种套接字种类有自己的通信寻址方法。Linux 将不同的地址族抽象统一为 BSD 套接字接口，应用程序关心的只是 BSD 套接字接口，通过参数来指定所使用的套接字地址族。

Linux 内核中为了支持多个地址族，定义了这么一个变量：static struct net_proto_family *net_families[NPROTO]，NPROTO 等于 32，也就是说 Linux 内核支持最多 32 种地址族。不过目前已经够用了，我们常用的不外乎就是 PF_UNIX（1）、PF_INET（2）、PF_NETLINK（16），Linux 还有一个自有的 PF_PACKET（17），即对网卡进行操作的选项。它们都通过如下的结构来定义，这个结构没有太多的成员：

```
1.  struct net_proto_family {
2.  int     family;     /* 这个值就是地址族的标识 */
3.  int     (*create)(struct socket *sock, int protocol);
4.  ......
5.  };
```

在 PF_INET 地址族之内，BSD 套接字还定义了多种我们熟知套接字类型，如流（stream），数据报（datagram），原始包（raw）等。

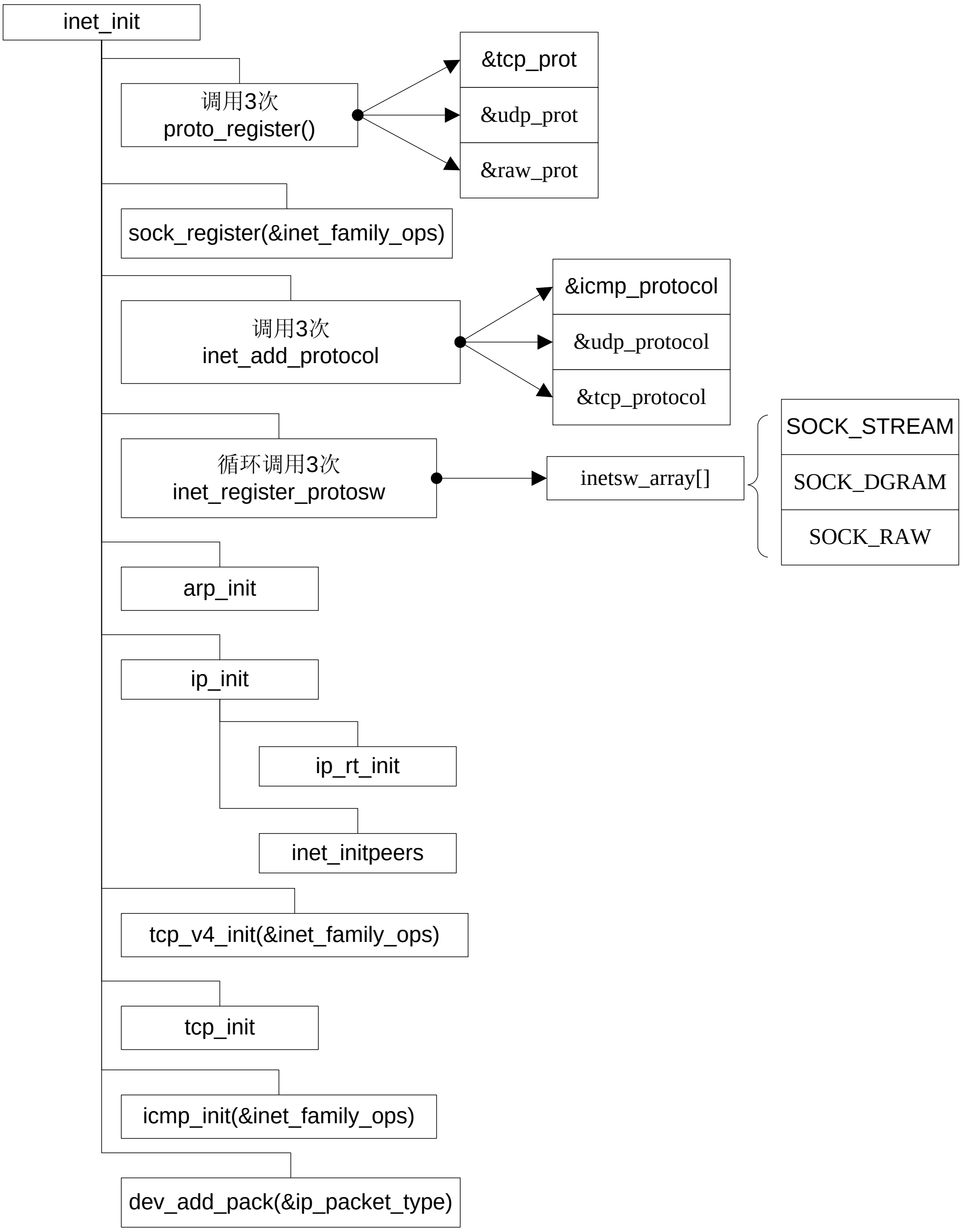
为了支持多种套接字类型，内核中是有多种相应的全局变量与之对应，而不是只有一种。比如 proto{}结构类型的，有 inet_protocol{}结构类型的，有 inet_protosw{}结构类型的，有 proto_ops{}结构类型的，四者的关系我会在下面介绍。

注意，网络协议的初始化是在网络设备的初始化之前完成的，在 Linux 系统中并不是说网络设备不存在就不需要网络协议了，而是在没有网络设备存在的时候，照样可以完成网络的工作，只不过网络系统物理上只存在于本机一台机器中而已。

tcp_v4_init( )和 tcp_init( )的不同：前者什么都不做（即不在本书的讨论范围内），而后者才是用来初始化 TCP 协议需要的各项 hash 表和 sysctl_xxx 全局配置项的。

arp_init 完成系统 neighbour 表的初始化。

ip_rt_init 初始化 IP 路由表 rt_hash_table，



图表 2-18 inet_init 调用树

一进入初始化就调用 proto_register3 次，先后为 tcp、udp、raw 的 proto{}结构申请空间并将其挂到一个全局链表 proto_list 上。这三个 proto 全局变量非常重要，是连接传输层和 IP 层的纽带。

```
1.  struct proto tcp_prot = {
2.  .name           = "TCP",
3.  .owner          = THIS_MODULE,
4.  .close          = tcp_close,
5.  .connect        = tcp_v4_connect,
6.  .accept         = inet_csk_accept,
7.  .ioctl          = tcp_ioctl,
8.  .init           = tcp_v4_init_sock,
```

```
9.  .sendmsg        = tcp_sendmsg,
10. .recvmsg        = tcp_recvmsg,
11. .backlog_rcv        = tcp_v4_do_rcv,
12. .hash           = tcp_v4_hash,
13. .get_port       = tcp_v4_get_port,
14. };
```

```
1.  struct proto udp_prot = {
2.      .name          = "UDP",
3.  .owner         = THIS_MODULE,
4.  .close         = udp_close,
5.  .connect       = ip4_datagram_connect,
6.  .disconnect    = udp_disconnect,
7.  .ioctl         = udp_ioctl,
8.  .sendmsg       = udp_sendmsg,
9.  .recvmsg       = udp_recvmsg,
10. .sendpage      = udp_sendpage,
11. .backlog_rcv       = udp_queue_rcv_skb,
12. .hash          = udp_v4_hash,
13. .unhash        = udp_v4_unhash,
14. .get_port      = udp_v4_get_port,
15. };
```

```
1.  struct proto raw_prot = {
2.  .name          = "RAW",
3.  .owner         = THIS_MODULE,
4.  .close         = raw_close,
5.  .connect       = ip4_datagram_connect,
6.  .disconnect    = udp_disconnect,
7.  .ioctl         = raw_ioctl,
8.  .init          = raw_init,
9.  .setsockopt    = raw_setsockopt,
10. .getsockopt    = raw_getsockopt,
11. .sendmsg       = raw_sendmsg,
12. .recvmsg       = raw_recvmsg,
13. .bind          = raw_bind,
14. .backlog_rcv       = raw_rcv_skb,
15. .hash          = raw_v4_hash,
16. .unhash        = raw_v4_unhash,
17. };
```

图表 2-19 tcp_prot, udp_prot, raw_prot 结构

再看 sock_register 函数，它把 inet_family_ops 塞入 net_families 数组中，这个 inet_family_ops 是如下定义的：

```
static struct net_proto_family inet_family_ops = {
    .family = PF_INET,
    .create = inet_create,
    .owner  = THIS_MODULE,
};
```

代码段 2-24 inet_family_ops 结构

以上的结构是应付从上到下方向的关系：用户创建 socket 时，先指定 INET 地址族，在指定套接字类型。换句话说这是数据流发送的流向。

下面这个结构是应付从下到上方向即数据流接收的关系。大家都知道，socket 层必须区分哪一个用户应该接收这个包，这叫做 socket 解复用。下面是初始化第二个方面的必要步骤：注册接收函数。

```
1.  /*
2.   *  Add a protocol handler to the hash tables
3.   */
4.  //在 inet_init 函数中调用了 3 次，分别传入 icmp_protocol，udp_protocol，tcp_protocol
5.  //这三个实体内容如下：
6.  /*
7.  static struct inet_protocol tcp_protocol = {
8.  .handler =  tcp_v4_rcv,
9.  .err_handler =  tcp_v4_err,
10. .no_policy =    1,
11. };
12. static struct inet_protocol udp_protocol = {
13. .handler =  udp_rcv,
14. .err_handler =  udp_err,
15. .no_policy =    1,
16. };
17. static struct inet_protocol icmp_protocol = {
18. .handler =  icmp_rcv,
19. };
20. */
21. int inet_add_protocol(struct inet_protocol *prot, unsigned char protocol)
22. {
23. int hash, ret;
24.
25. hash = protocol & (MAX_INET_PROTOS - 1);
26.
27. if (inet_protos[hash]) {
28.     ret = -1;
29. } else {
30.     inet_protos[hash] = prot;
31.     ret = 0;
32. }
33.
34. return ret;
35. }
```

```
struct inet_protocol
{
    int (*handler)( sk_buff *skb);
    void(*err_handler)(sk_buff*skb,u32 info);
    int no_policy;
};
```

代码段 2-25inet_add_protocol 函数

在这段代码相关的内容中，我们特别要记住那三个结构的接收函数指针：tcp_v4_rcv，udp_rcv，icmp_rcv 我们将在数据接收过程中详细介绍这几个函数。

Linux 区分永久和非永久协议。永久协议包括象 UDP 和 TCP，这是 TCP/IP 协议实现的基本部分，去掉一个永久协议是不允许的。所以，UDP 和 TCP 是不能 unregistered。此机制由 2 个函数和一个维护注册协议的数据结构组成。一个负责注册协议，另一个负责注销。每一个注册的协议都放在一个表里，叫协议切换表。表中的每一个入口是一个 inet_protosw 的实例。先看基于这个结构产生的三个实例：

```
1.  /* Upon startup we insert all the elements in inetsw_array[] into
2.   * the linked list inetsw.
3.   */
4.  static struct inet_protosw inetsw_array[] =
5.  {
6.         {
7.                .type =      SOCK_STREAM,
8.                .protocol =   IPPROTO_TCP,
9.                .prot =      &tcp_prot,
10.               .ops =       &inet_stream_ops,
11.               .capability = -1,
12.               .no_check =   0,
13.               .flags =     INET_PROTOSW_PERMANENT,
14.        },
15.
16.        {
17.               .type =      SOCK_DGRAM,
18.               .protocol =   IPPROTO_UDP,
19.               .prot =      &udp_prot,
20.               .ops =       &inet_dgram_ops,
21.               .capability = -1,
22.               .no_check =   UDP_CSUM_DEFAULT,
23.               .flags =     INET_PROTOSW_PERMANENT,
24.       },
25.
26.       {
27.              .type =      SOCK_RAW,
28.              .protocol =   IPPROTO_IP,   /* wild card */
29.              .prot =      &raw_prot,
30.              .ops =       &inet_dgram_ops,
31.              .capability = CAP_NET_RAW,
32.              .no_check =   UDP_CSUM_DEFAULT,
33.              .flags =     INET_PROTOSW_REUSE,
34.       }
35. };
```

```
struct inet_protosw {
    struct list_head list;

   两个字段形成查找 key
   socket 系统调用用的第二个参数.
    ushort type;
    int protocol; /* L4 的协议号  */
    struct proto     *prot;
    struct proto_ops *ops;
    int      capability;
  发送或接收报文时是否需要校验和
    char    no_check
    uchar    flags;
};
```

代码段 2-26 inet_protosw 结构

也就是说在 inet_init 中循环调用了 inet_register_protosw 三次。传入的参数就是上面列出的 inetsw_array[ ]的每个单元。分别对应 TCP、UDP、RAW 协议。这 inetsw 将来会在创建 socket 的时候用到。

```
1.  #define INETSW_ARRAY_LEN (sizeof(inetsw_array) / sizeof(struct inet_protosw))
2.
3.  void inet_register_protosw(struct inet_protosw *p)
4.  {
5.  struct list_head *lh;
6.  struct inet_protosw *answer;
7.  int protocol = p->protocol;
8.  struct list_head *last_perm;
9.
10. ......
11.
12. /* If we are trying to override a permanent protocol,
13. answer = NULL;
14. last_perm = &inetsw[p->type];
15. list_for_each(lh, &inetsw[p->type]) {
16.     answer = list_entry(lh, struct inet_protosw, list)
17.
18.     /* Check only the non-wild match. */
19.     if (INET_PROTOSW_PERMANENT & answer->flags) {
20.         if (protocol == answer->protocol)
21.             break;
22.         last_perm = lh;
23.     }
24.
25.     answer = NULL;
26. }
```

在缺省初始化时只有 inetsw_array[i]，但如果加入了 SCTP，那么就加入了 sctp_seqpacket_protosw

```
enum sock_type {
        SOCK_STREAM = 1,
        SOCK_DGRAM  = 2,
        SOCK_RAW    = 3,
        ......
        SOCK_PACKET = 10,
};

#define SOCK_MAX (SOCK_PACKET + 1)
struct list_head inetsw[SOCK_MAX];
```

```
27. if (answer)
28.     goto out_permanent;
29. list_add_rcu(&p->list, last_perm);
30. out:
31.
32. synchronize_net();
33.
34. return;
35. …….//错误处理，退出
36. }
```

代码段 2-27　inet_register_protosw 函数

现在研究 **inet_stream_ops，inet_dgram_ops** 的结构：

```
1.  struct proto_ops inet_stream_ops = {
2.      .family =   PF_INET,
3.       .release = inet_release,
4.      .bind =     inet_bind,
5.      .connect =  inet_stream_connect,
6.      .socketpair =   sock_no_socketpair,
7.      .accept =   inet_accept,
8.      .poll =     tcp_poll,
9.      .ioctl =    inet_ioctl,
10.     .listen =   inet_listen,
11.     .shutdown = inet_shutdown,
12.     .setsockopt =   inet_setsockopt,
13.     .getsockopt =   inet_getsockopt,
14.     .sendmsg =  inet_sendmsg,
15.     .recvmsg =  inet_recvmsg,
16.     .mmap =     sock_no_mmap,
17.     .sendpage = tcp_sendpage
18.     };
19.
20. struct proto_ops inet_dgram_ops = {
21.     .family =   PF_INET,
22.      .release = inet_release,
23.     .bind =     inet_bind,
24.     .connect =  inet_dgram_connect,
25.     .socketpair =   sock_no_socketpair,
26.     .accept =   sock_no_accept,
27.     .getname =  inet_getname,
28.     .poll =     datagram_poll,
29.     .ioctl =    inet_ioctl,
30.     .listen =   sock_no_listen,
31.     .shutdown = inet_shutdown,
32.     .setsockopt =   inet_setsockopt,
33.     .getsockopt =   inet_getsockopt,
34.     .sendmsg =  inet_sendmsg,
35.     .recvmsg =  inet_recvmsg,
36.     .mmap =     sock_no_mmap,
37.     .sendpage = inet_sendpage,
38. };
39. /*
40.  * 对于 SOCK_RAW sockets，除了没有 udp_poll 外其它必须和 inet_dgram_ops 一样
41.  */
42. static const struct proto_ops inet_sockraw_ops = {
43.     .family        = PF_INET,
44.     .owner         = THIS_MODULE,
45.     .release       = inet_release,
46.     .bind          = inet_bind,
47.     .connect       = inet_dgram_connect,
48.      ......
49. };
```

```
struct proto_ops {
    int family;

    int (*release) (socket *sock);
    int (*bind)  (socket *sock, sockaddr *myaddr, int sockaddr_len);
    int (*connect) (socket *sock, sockaddr *vaddr, int sockaddr_len, int flags);
    int (*accept) (socket *sock, socket *newsock, int flags);
    uint (*poll) (file *file, socket *sock, poll_table_struct *wait);
    int (*ioctl) (socket *sock, uint cmd,  ulong arg);
    int (*listen) (socket *sock, int len);
     int(*sendmsg) (kiocb *iocb, socket *sock, msghdr *m, size_t total_len);
    int (*recvmsg) (kiocb *iocb, socket *sock, msghdr *m, size_t total_len, int flags);
     ssize_t(*sendpage) (socket *sock, page *page, int offset, size_t size, int flags);
};
```

代码段 2-28　inet_stream_ops，inet_dgram_ops，inet_sockraw_ops 结构

以上是关于协议栈框架的搭建，看起来似乎完美了，但是对于 IP 层接收流程，则还不够。因为对于发送过程，直接调用 IP 层函数，而对于内核接收过程则分为 2 层：上层需要有一个接收函数解复用传输协议报文，我们已经介绍了，而下层需要一个接收函数解复用网络层报文。对报文感兴趣的底层协议目前有两个，一个是 ARP，一个是 IP，报文从设备层送到上层之前，必须区分是 IP 报文还是 ARP 报文。然后才能往上层送。这个过程由一个数据结构来抽象，叫 packet_type{}，

```
1.  struct packet_type {
2.      unsigned short  type;   /* This is really htons(ether_type).   */
         dev 是指向我们希望接收到包的那个接口的 net device 结构。如果是 NULL，则我们会从任何一个网络
         接口上收到包
3.      struct net_device   *dev;   /* NULL is wildcarded here      */
4.      int  (*func) (struct sk_buff *, struct net_device *, struct packet_type *);
5.      void    *af_packet_priv;
        如果某个 packet_type{}被注册到系统中，那么它就被挂接到全局链表中（有 2 个，见下面的解说），
        list 就是代表链表节点
6.      struct list_head    list;
7.  };
```

代码段 2-29　packet_type 结构定义

inet_init 函数最后调用了一个 dev_add_pack 函数，不仅是 inet_init 函数调用，有一个很很重要的模块也调用了它，就是 ARP 模块，我们会在后面的章节看到它是如何调用 dev_add_pack 函数的。

```
1.  /*
2.   *  For efficiency
3.   */
4.  
5.  int netdev_nit;
6.  
7.  /**
8.   *  dev_add_pack - add packet handler
9.   *  @pt: packet type declaration
10.  *
11.  *  Add a protocol handler to the networking stack. The passed &packet_type
12.  *  is linked into kernel lists and may not be freed until it has been
13.  *  removed from the kernel lists.
14.  *
15.  *  This call does not sleep therefore it can not
16.  *  guarantee all CPU's that are in middle of receiving packets
17.  *  will see the new packet type (until the next received packet).
18.  */
19. 
20. void dev_add_pack(struct packet_type *pt)
21. {
22.     int hash;
23. 

        ......

        对于 arp 和 ip 报文都不是 ETH_P_ALL，所以挂到了 ptype_base 这个 hash 表中。
24.     if (pt->type == htons(ETH_P_ALL)) {
25.         netdev_nit++;
26.         list_add_rcu(&pt->list, &ptype_all);
27.     } else {
28.         hash = ntohs(pt->type) & 15;
29.         list_add_rcu(&pt->list, &ptype_base[hash]);
30.     }
31. }
```

```
static struct packet_type arp_packet_type = {
    .type = __constant_htons(ETH_P_ARP),
    .func = arp_rcv,
};
```

```
static struct packet_type ip_packet_type = {
    .type = __constant_htons(ETH_P_IP),
    .func = ip_rcv,
};
```

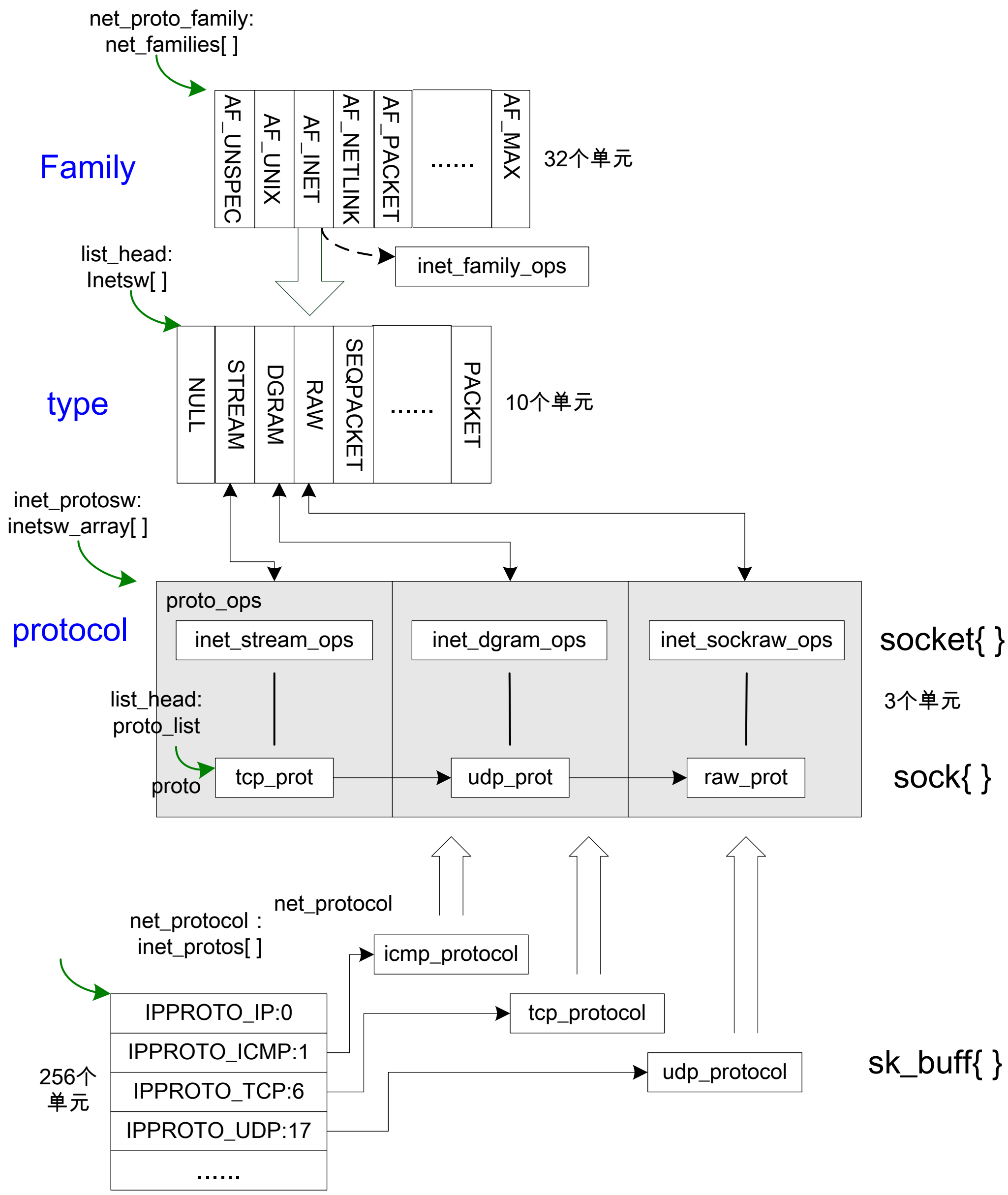
代码段 2-30　dev_add_pack 函数

报文处理函数放置在一个链表中，这是因为多个协议可能为同一种包注册不同的处理函数。由于多个协议可能接收同一种包，所以它们在复制包之前不能修改其中的内容。在 Linux 内核中 IP 协议栈很少

有协议是 ETH_P_ALL 类型的，即接收所有报文，因为这会影响转发性能。

传说中 Linux 有一个能抓包的特性，方法如下：fd = socket(PF_PACKET, mode, htons(ETH_P_ALL)); 如此这般就能把所有的报文收上来。实际上内核就是把一个 type 为 ETH_P_ALL 的 prot_hook 挂接到 ptype_all 链表上，其报文处理函数为 packet_rcv，感兴趣的读者可以在研究完 socket 系统调用的章节后再去看 af_packet.c 中的 packet_create 函数。

综合前面分析的初始化过程，那么网络协议栈的框架基本搭起来了，从上图中可以看到代码中用数个全局变量完成了 INET 层和传输层的搭建工作。记住，在这里都还只是 INET 层和传输层的组织架构，而 IP 层则没有全局变量去代表，只有函数，我们将在以后的过程中对其解剖。



图表 2-20 协议栈的具体形式

综上所述，我们可以推断出在本节初始化的过程中出现的各种重要的数据结构之间的关系了，正如上图所示，从左往右看，是用户界面的角度，分别代表了标识一个套接字的三元组：<地址族，类型，具体协议>，正好是调用 socket 系统函数的 3 个参数。通过内核中这 3 个数据结构，我们就可以创建 sock{}结构。从右到左看，是内核中 3 个重要的数据结构，从上到下分别是 socket{}、sock{}、sk_buff{}，正好是数据流的连接通道。比如，发送报文时，数据会由 socket{}通过相应的 proto_ops{}把数据传给 sock{}，sock{}又通过 proto{}把数据传到 sk_buff；反过来，当收到报文时，sk_buff{}通过 net_protocol{}把数据传

给 sock{}，后者又通过 proto{}把数据传给 socket{}，socket{}最后把数据传给用户层。通过这几个重要的数据结构和变量，大家是不是看出了所谓“栈”的模式？

所以，这几个数据结构结合非常紧密，不过比较恶心的是它们的名字比较容易混淆，所以说，开发 Linux 的内核源代码要小心啊。

2.5.5 初步了解路由系统

Linux 内核中采用了 FIB（Forward Information Base）这个名词代替了 Routing Database，原因不详。可能是不想和应用层的路由数据库发生概念上的冲突吧。但是 Linux 内核还是有一个叫做 RouteTable 的数据结构的，不过，它只是 FIB 的一份 cache 而已，其关系如同计算机中内存和 CPU cache 的关系。系统中路由一般采取的手段是：先到路由缓存中查找表项，如果能查找到，那么就直接将对应的一项取出作为路由的规则；如果查不到，那么就到 FIB 中根据规则换算出来，并且增加一项新的，在路由缓存中将项目添加进去。所以在研究 Linux 代码时，应该注意这一点，不能抓着 RouteTable 不放而忽视了 FIB。

FIB 是内核中最重要的路由结构。FIB 存放着用来给本地流量和外发报文做内部的路由，也能让内核外的应用程序通过路由 socket 从内核中获取路由信息。本质上它是一个表，包含上层的地址信息和底层的设备信息。通过对 FIB 数据的查找和换算，一定能够获得路由一个地址的方法。当报文进入路由系统时，系统用报文的目标地址和最精确的网络掩码比较，如果不匹配，就转到另一个较精确的掩码入口和其比较。当完成比较后，IP 层复制到远地主机的“direction”到路由快表中，并沿着这条路径发送数据。

Linux 能被配置成支持多个 FIB/路由表，在这里，FIB 表就是路由表，一个 FIB 表包含多个路由条目，以下的语境中，你可以把“FIB 表”当作一个数据库的代名词。缺省情况下只配置有 2 个表（即 2 个数据库）。在大多数情况下，Linux 内核不需要基于策略的路由，特别是嵌入式系统。所以内核不需要配置多个表。如果这样，有两个预定义的全局指针指向两个表，一个是 local table, 一个是 main table，local 表存放着到本机分配的地址的路由上，比如分配给网络接口设备的地址和环回地址，main 表存放到外部节点的路由。

由于这个系统相当复杂，我们将这部分的初始化放到下一章去介绍，在此时，只需记住了 FIB 表就是路由表，Linux 的路由表就是 FIB 表。还有，是 ip_init 函数调用了 ip_rt_init 去初始化路由系统。

2.6 Linux 设备管理

设备初始化是我们要分析的第三和第四个大步骤，这个部分要涉及到一些设备驱动的背景知识。

设备管理的目标是能对所有的外设进行良好的读、写、控制等操作。但是如果众多设备没有一个统一的接口，则不利于开发人员的工作。因此 Linux 采用了类似 UNIX 的方法，使用设备文件来实现这个统一接口。由此可见，设备文件的相关概念是设备管理的最基础部分。

要让操作系统感知到设备的存在，必须提供一个注册机制，使操作系统能识别设备对应的驱动程序，目前采用基于主、次设备号的方式来管理设备。Linux 习惯上将设备文件放在目录/dev 或其子目录之下，设备文件名通常由两部分组成，第一部分通常较短，可能只有 2 或 3 个字母组成，例如普通硬盘如 IDE 接口的为“hd”，SCSI 硬盘为“sd”，软盘为“fd”，并口为“lp”，第二部分通常为数字或字母，用来区别设备实例。例如/dev/hda，/dev/hdb，/dev/hdc 表示第一、二、三块硬盘；而/dev/hda1、/dev/hda2、/dev/hda3 表示第一硬盘的第一、第二、第三分区。

这种机制就是 2.4.的版本中的注册与管理方式——devfs 管理方式，如果在编译内核时选中 CONFIG_DEVFS_FS，那么就可以利用这种设备管理方式。devfs 挂载于/dev 目录下，提供了一种类似于文件的方法来管理位于/dev 目录下的所有设备，我们知道/dev 目录下的每一个文件都对应的是一个设备，至于当前该设备存在与否先且不论，而且这些特殊文件是位于根文件系统上的，在制作文件系统的时候我们就已经建立了这些设备文件，因此通过操作这些特殊文件，可以实现与内核进行交互。但是 devfs 文件系统有一些缺点，例如：不确定的设备映射，有时一个设备映射的设备文件可能不同，例如我的 U 盘可能对应 sda 有可能对应 sdb；没有足够的主/辅设备号，当设备过多的时候，显然这会成为一个问题；/dev 目录下文件太多而且不能表示当前系统上的实际设备；命名不够灵活，不能任意指定，容易造成设备之间冲突而不能正常初始化驱动。

本人在构思这本书的时候 devfs 管理方式还在大行其道，但是时隔 3 年，内核源代码中已经不采用这种技术，转而采用 sysfs 技术。引入了一个新的文件系统 sysfs，它挂载于/sys 目录下，跟 devfs 一样它也是一个虚拟文件系统，也是用来对系统的设备进行管理的，它把实际连接到系统上的设备和总线组织成

一个分级的文件，用户空间的程序同样可以利用这些信息以实现和内核的交互，该文件系统是当前系统上实际设备树的一个直观反应，*sysfs* 的工作就是把系统的硬件配置视图导出给用户空间的进程。在 2.6.18 内核中，必须选中 General Setup→Configure Standard Kernel features（For small systems），在 File System→Pseudo filesystems 菜单内部出现“sysfs file system support”。

不管如何进步，其中心思想是建立一种分层的体制，块设备是一种 class，字符设备是一种 class，而网络设备也是一种 class，驱动程序开发者如果知道自己的设备属于哪一种 class，那么就把其驱动程序挂到相应的 class 上，让内核为驱动程序分配名字和设备号，如果不确定是哪种 class，还可以自己建立 class 类别。Linux 用户可以到/sys 下观察一下系统中有哪些内核模块及驱动，然后和/dev 下的文件做一个对比就发现，/sys 目录确实将各种设备进行了归类，条理清晰的多。用户空间的工具 udev 就是利用了 sysfs 提供的信息来实现所有 devfs 的功能的，但不同的是 udev 运行在用户空间中，而 devfs 却运行在内核空间，而且 udev 不存在 devfs 那些先天的缺陷。很显然，sysfs 将是未来发展的方向。

那么 sysfs 是怎么认出系统中存在的设备以及应该使用什么设备号呢？对于已经编入内核的驱动程序，当被内核检测到的时候，会直接在 sysfs 中注册其对象；对于编译成模块的驱动程序，当模块载入的时候才会这样做。一旦挂载了 sysfs 文件系统(挂载到 /sys)，内建的驱动程序在 sysfs 注册的数据就可以被用户空间的进程使用，并提供给 udev 以创建设备节点。

关于设备管理的内容，我只想说这么多，因为要牵扯到许多的配置文件和环境变量，由于每个版本的 Linux 的设备管理在配置文件和环境变量的设置上多少有些不同，它们的进化又非常快，我觉得只要不影响本文的理解，我们就先跨过这部分内容吧。下面我们开始进入内核设备管理系统。

Linux 在设备驱动程序的实现上又分为两层：

- 抽象设备层（又叫核心模块）
- 特定设备驱动程序

抽象硬件层：

这一层主要提供一些设备无关的处理流程，也提供一些公用的函数给底层的 device driver 调用。它为网络协议提供统一的发送、接收接口。这主要是通过 net_device 结构。是上层的、与设备无关的，这部分根据输入输出请求，通过特定设备驱动程序接口，来与设备进行通信。

特定设备驱动程序：是一种下层的、与设备有关的，常称为设备驱动程序，它直接与相应设备打交道，并且向上层提供一组访问接口； 当一个网络设备的初始化程序被调用时，它返回一个状态指示它所驱动的控制器是否有一个实例。

那么第三个大步骤就是抽象设备层的初始化。这由 net_dev_init 函数完成，此函数由下面的宏修饰，如下：

subsys_initcall(net_dev_init);

这个宏定义请参见前面说的 init.h，它被定义为：define_initcall("4",fn)

所以它是在 core_initcall 和 fs_initcall 之后被调用的。

在 Linux2.4 内核中 net_dev_init 就是对实际底层网络设备的初始化例程。但是我们要注意的是现在的这个函数在 Linux2.6 内核中已经不对特定设备进行初始化了，只是为网络设备设置一些基础功能。比如 proc 文件系统、sysfs 系统、全局设备和索引表、设置软中断回调等。不过我个人觉得最重要的是对 queue 的各项成员的初始化。

```
1.  /*
2.   *  初始化可以存放设备相关信息的全局数据结构，在此时设备一般还没有被初始化，
3.   * 此函数由单一线程在 boot 的时候调用，不需要获取 rtnl 信号量
4.  */
5.  static int __init net_dev_init(void)
6.  {
7.      int i, rc = -ENOMEM;
8.  
9.      BUG_ON(!dev_boot_phase);
```

```
10.
11.     if (dev_proc_init())
12.         goto out;
13.
14.     if (netdev_sysfs_init())
15.         goto out;
16.
17.     INIT_LIST_HEAD(&ptype_all);
18.     for (i = 0; i < 16; i++)
19.         INIT_LIST_HEAD(&ptype_base[i]);
20.
21.     for (i = 0; i < ARRAY_SIZE(dev_name_head); i++)
22.         INIT_HLIST_HEAD(&dev_name_head[i]);
23.
24.     for (i = 0; i < ARRAY_SIZE(dev_index_head); i++)
25.         INIT_HLIST_HEAD(&dev_index_head[i]);
26.
27.     /*
28.      *  Initialise the packet receive queues.
29.      */
    每个 CPU 有自己的一个队列，本书不打算讨论多 CPU 的情形，所以读者请将 NR_CPUS 看成 1 即可
30.     for (i = 0; i < NR_CPUS; i++) {
31.         struct softnet_data *queue;
32.         queue = &per_cpu(softnet_data, i);
33.         skb_queue_head_init(&queue->input_pkt_queue);
34.         queue->throttle = 0;
35.         queue->cng_level = 0;
36.         queue->avg_blog = 10; /* arbitrary non-zero */
37.         queue->completion_queue = NULL;
38.         INIT_LIST_HEAD(&queue->poll_list);
39.         set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
40.         queue->backlog_dev.weight = weight_p;
41.         queue->backlog_dev.poll = process_backlog;
42.         atomic_set(&queue->backlog_dev.refcnt, 1);
43.     }
44.
45.     dev_boot_phase = 0;
46.
47.     open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
48.     open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
49.
50.     dst_init();
51.     dev_mcast_init();
52.     rc = 0;
53.     out:
54.     return rc;
55. }
```

每个 CPU 把报文放到自己的队列中，所以不需要 lock：

```
struct softnet_data
{
    struct net_device   *output_queue;
    struct sk_buff_head input_pkt_queue;
    struct list_head    poll_list;
    struct sk_buff  *completion_queue;

    struct net_device   backlog_dev;
};
```

有一个全局变量叫 per_cpu__softnet_data，定义为 CPU 软中断（包括接收和发送）的数据队列，这个变量是通过 DEFINE_PER_CPU(struct softnet_data, softnet_data) = { NULL };定义的

```
void __init dst_init(void)
{
    register_netdevice_notifier(&dst_dev_notifier);
}
```

其中的参数是通知链 dst_dev_notifier，它在初始化过程中没有什么特别重要的用处，但是在删除设备接口的时候非常重要，因为它只响应删除事件

代码段 2-31 net_dev_init 函数

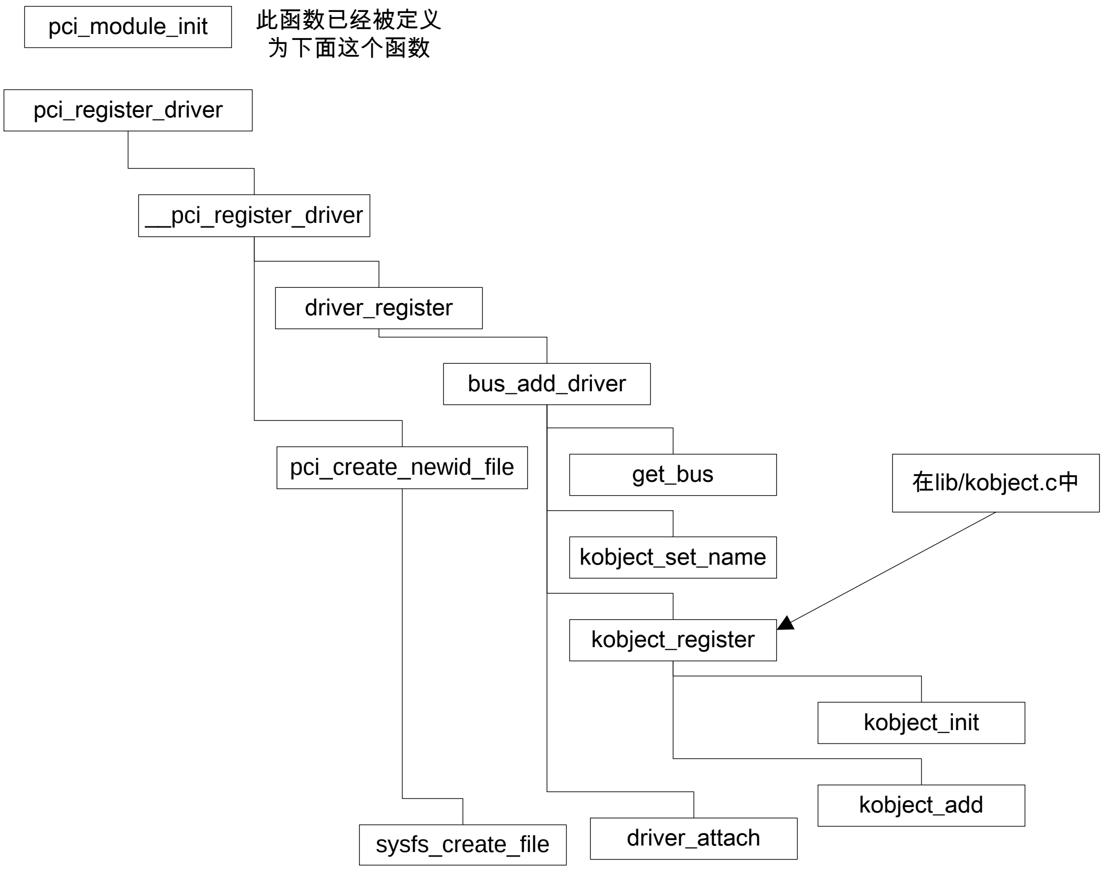
这个 queue 之所以重要，在于我们将来在接收报文的章节中要对它大书特书，不然，我们就不知道 Linux 网络接收过程的整个环节，不过，在此处，读者只需记住 2 点：

1. 这个 queue 有一个名叫 backlog_dev 的设备，其 poll 函数指针指向了一个叫做 process_backlog 的函数
2. 我们设置的接收软中断(RX_SOFTIRQ)将要和这个 queue 以及 back_log 设备打交道。

2.6.1 底层 PCI 模块的初始化

虽然我们不必太关心设备管理是如何实现的，但是我们要知道我们的驱动程序是如何与设备搭上关系。在目前的主机上，PCI 总线是用得最广泛的总线技术，而基于 PCI 总线的网卡设备已经是市场主流，我们就研究一下 PCI 网卡是如何被操控的，以此可以推断出在不同总线技术下驱动程序的实现基础。

我们已经知道万事都得有头，驱动程序也不例外。在之前提到的 4 个大的步骤里，驱动程序开发人员使用 module_init 宏来修饰自己驱动程序的第一个函数，促使初始化函数放在第 6 段.initcall 段中（如果你忘记了请参考 2.2.3 节和 2.5 节）。驱动程序必须得遵守一套开发框架，如果是 PCI 驱动，那么为了做到这一点必须调用一个函数：pci_module_init，它就是整个驱动程序开始工作的第一步，它也能替你完成底层关于 PCI 操作。不过在 2.6 中它已经变成 include/linux 目录下的 pci.h 中的宏，它被定义成 pci_register_driver。其参数是一个 pci_driver{}类型的结构体，此结构由驱动开发人员定义。我们等一会还能见到它。



图表 2-21 pci_module_init 函数调用树

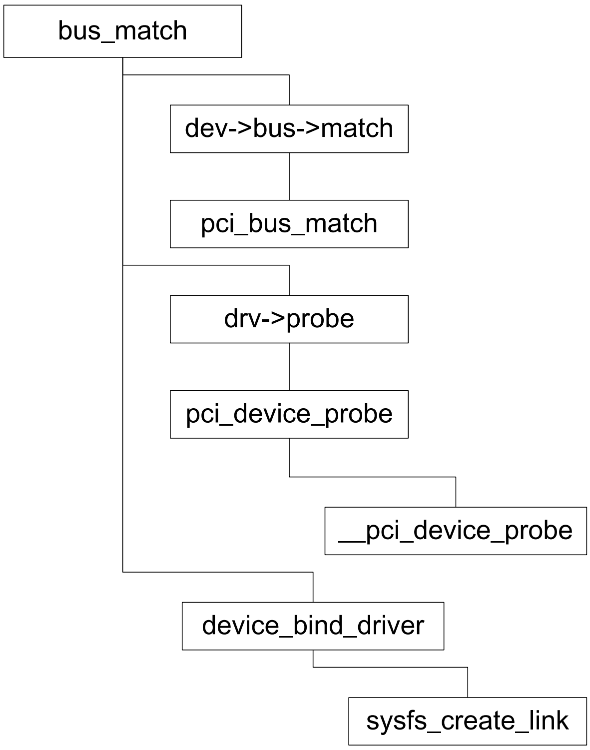
根据上图顺藤摸瓜，最终会找到 driver_attach 函数。先仔细分析它：

```
1.  /**
2.   *  driver_attach – 尝试着把驱动绑定到设备上
3.   *  @drv:   代表驱动程序的结构.
4.   *
5.   *  遍历总线上挂着的设备链表，并尝试匹配每个驱动，如果 bus_macth()返回 0 并且 dev->driver 被设置
    了，那么也就找到了匹配对。注意，我们忽略返回值为-ENODEV 的结果，因为很有可能一个驱动程序不用绑定到
    一个设备上，这种情况其实很常见，主要是某些需求要求运行在内核，但又没有实际的设备与之对应，人们以内核
    模块的方式将这种代码植入内核——比如病毒。
6.   */
7.  void driver_attach(struct device_driver * drv)
8.  {
9.      struct bus_type * bus = drv->bus;
10.     struct list_head * entry;
11.     int error;
12. 
13.     if (!bus->match)
14.         return;
       遍历整个总线链表，找到与设备匹配的驱动程序，这说明总线上挂的不止一种设备
15.     list_for_each(entry,&bus->devices.list) {
16.         struct device * dev = container_of(entry,struct device,bus_list);
17.         if (!dev->driver) {
```

```
18.             error = bus_match(dev,drv);
19.             ......
20.         }
21.     }
22. }
```

代码段 2-32　driver_attach 函数

我们就没有必要去翻箱倒柜的研究 bus_macth 内部做了什么了，要把它说明白就偏离了本书的主线。我们只给出这个函数的大致流程。



图表 2-22 bus_match 函数调用树

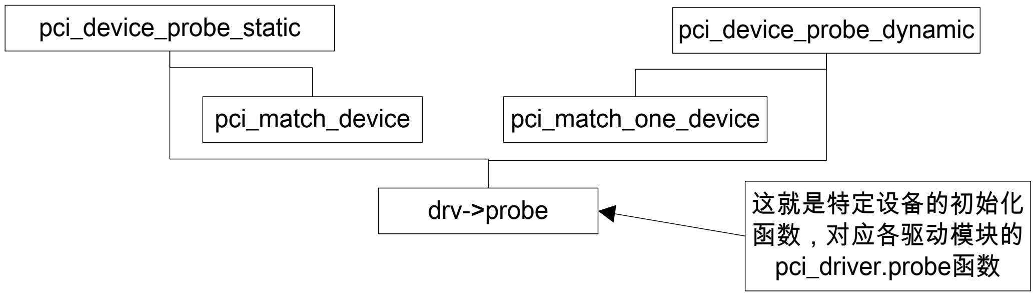
bus_match 函数最后会调用__pci_device_probe，这个函数非常重要，和所有驱动程序有非常大的关系：

```
1.  /**
2.   * 如果成功就返回 0，否则返回非 0 值.
3.   * side-effect: pci_dev->driver is set to drv when drv claims pci_dev.
4.   */
5.  static int
6.  __pci_device_probe(struct pci_driver *drv, struct pci_dev *pci_dev)
7.  {
8.      int error = 0;
9.
10.     if (!pci_dev->driver && drv->probe) {
11.         error = pci_device_probe_static(drv, pci_dev);
12.         if (error == -ENODEV)
13.             error = pci_device_probe_dynamic(drv, pci_dev);
14.     }
15.     return error;
16. }
```

代码段 2-33　__pci_device_probe 函数

不管是哪一种 probe，都最终调用 drv→probe，这是一个函数指针，它是由驱动程序的开发者设置的。下面就以网络设备的驱动程序为例，看看此 probe 是如何被指定的，它完成了什么工作。下图表示 dev→probe 可以被两个函数调用，分别是 pci_device_probe_static 和 pci_device_probe_dynamic 函数，

它们先分别调用 pci_match_device 和 pci_match_one_device 函数，最后才会调用它。



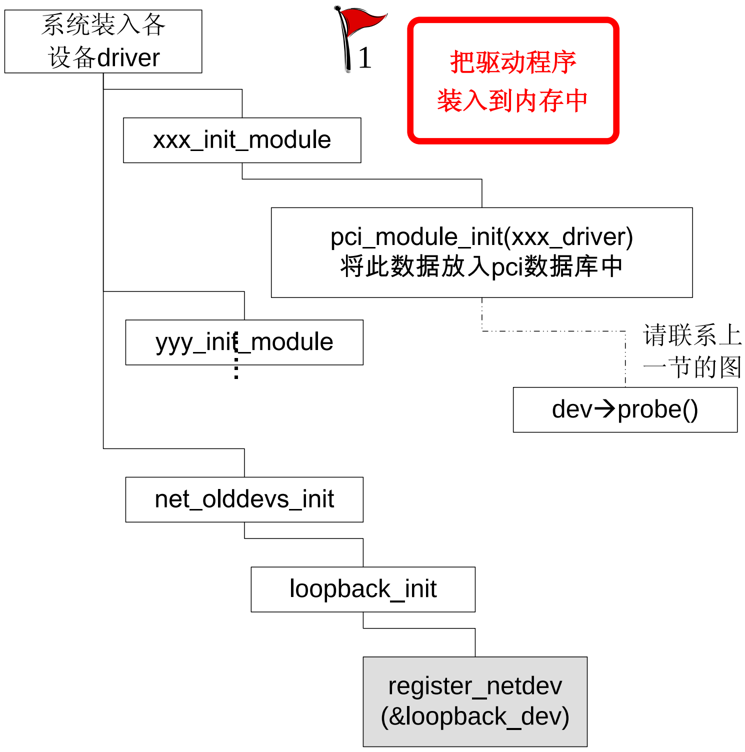
图表 2-23 drv->probe 的被调用关系树

在__pci_register_driver 调用 driver_register 之后会调用 pci_create_newid_file 函数，它会调用 sysfs_create_file 函数为当前的驱动程序创建相应的文件，文件放在/sys 目录下。请参考前文所述。

2.6.2 网络设备接口初始化例程

net_dev_init 为我们准备好了网络设备的基础功能部件，那特定的设备驱动程序在哪被初始化呢？

先别急，我们得先知道驱动程序是如何被装入内存的。我们上面章节曾分析过 ELF 格式，每个驱动程序编写者通过设置自己的驱动程序入口函数——比如 xxx_init_module——为 module_init 类型的，那么在编译以后入口函数会放在特殊的 text 段中以至于系统在启动的时候能找到这个入口函数，就这样，系统遍历并执行各个入口函数，让这些驱动程序完成基本的加载。下图中就是驱动程序被装入内存中的实现过程，其中包括我们熟知的 loopback 接口，它也作为一种设备，在这个阶段被装入内存，而且，注意，它还主动完成了其它驱动程序没有做的事——register_netdev！：



图表 2-24 系统装入各驱动程序的步骤

设备驱动程序被装入系统的步骤如上图所示。每个驱动程序调用 pci_module_init 函数，传入一个 pci_driver{}结构，比如：

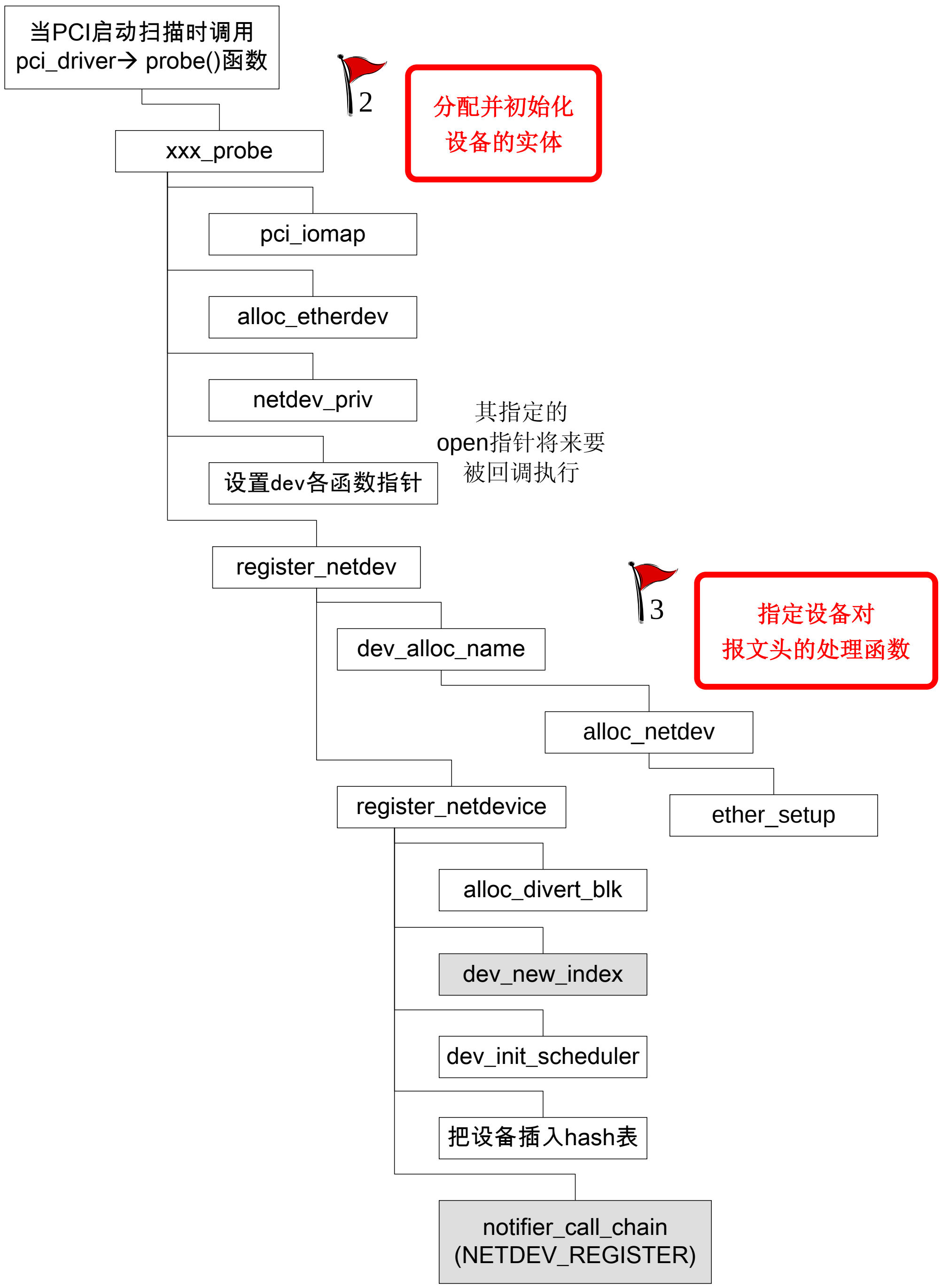
```
static struct pci_driver aaa_driver = {
	.name	= "aaa",		/* 驱动模块的名字，可任由开发者定义 */
```

```
    .id_table = aaa_pci_tbl,    /* 这是一个 pci_device_id{}结构的数组，必须和设备的硬件信息一致 */
    .probe    = aaa_init_one,   /* 回调函数，由 PCI 模块调用 */
};
```

不过要注意的是这只是加载，并不是对设备进行初始化。在 Linux 初学者会有错觉。一个明证就是：我们在配置内核的编译时，比如在配置网络设备一节时，会看到多种网络设备驱动被编译到内核中，但是实际上能起作用的就只是与主机网口真正匹配的驱动程序能工作。也就是说，那些没有相应设备的驱动程序根本就找不到自己的设备，也就无所谓“初始化”了。举个例子，本人主机网卡是 Ether ExpressPro100，但在在配置编译选项时，如果选择同时编译 Ether ExpressPro100 和 Intel Pro /100+并都是内嵌入内核时，系统会先初始化前者，而后者只执行到 pci_bus_match 就返回了 0——没有找到相应的设备!。

那么真正的初始化在哪呢？

结合上一节介绍的底层 PCI 模块的初始化一节中，我们知道每个驱动程序必须设置代码中的 device id{}结构。驱动程序把这个结构传入 PCI 数据库，当 PCI 开始工作以后，它会扫描总线和设备的 device id，然后查找数据库中的每个驱动，如果与之匹配，那么就会调用之前注册到 PCI 库中的 dev→probe（）函数。每个驱动程序要自己写 probe（）回调函数，完成检查寄存器和真正初始化设备的工作。其通用的工作流程如下图：



图表 2-25 drv→probe 实现的基本功能

- dev_new_index 中分配一个 ifindex 给设备，所谓分配，其实就是一个 static 整数不断往上加 1，来一个设备就加 1。
- 在 2.6 早期版本中用 dev_base 数组来串联每个设备，但是这种方法在查找上不方便，而且扩展性差（当时只有 8 个设备），随着 Linux 在路由器和交换机设备上的应用，这种方法不能适应这种类型设备多接口的特性，转而采用 dev_name_head 来记录。此全局变量是一个 hash 表，以设备的名字作为输入，再通过一系列 hash 变换得到一个 key，来查找和增删某设备，如果碰到接口特别多的情况比如有 128 个接口，或者有 4095 个 VLAN，这样的 hash 查找就比较快了。（Linux 下 VLAN 也是一种特殊的“设备”），当然系统中还有使用 dev_base 的地方，比如要搜索一个接口但并不知道其名字和接口，只知道接口的 ip 地址，那么就只好从这个表找——咦，这不就是路由查找的本质吗？
- static struct hlist_head dev_name_head[1<<NETDEV_HASHBITS]，也就是说该 hash 数组有 15 个

单元，比以前的 8 个多，如果发生 key 冲突可以用链表来挂接。

- 同样的道理，网口的 ifindex 也用 hash 表来存储了，名字叫 dev_index_head

本来不想列出函数的样子，后来想想如果不详细列出来，那么以后很多事情讲不清楚。

```
1.  /**
2.   *  register_netdevice  - 注册一个网络设备
3.   *  @dev: 是将要创建的设备指针，我们后面会详细给出它的结构定义
4.   *
    此函数把一个网络设备加到内核中，然后发送一个 NETDEV_REGISTER 消息给 netdev 通知链， 如果成返回 0，
    如果失败就返回一个负的错误号， 指示其出现什么样的错误，你也可以调用  register_netdev() 代替这个
    函数
5.  */
6.
7.  int register_netdevice(struct net_device *dev)
8.  {
9.  struct hlist_head *head;
10. struct hlist_node *p;
11. int ret;
12.
13. /* When net_device's are persistent, this will be fatal. */
14. BUG_ON(dev->reg_state != NETREG_UNINITIALIZED);
15. ......
16. dev->iflink = -1;
17.
18. /* Init, if this function is available */
19. if (dev->init) {
20.     ret = dev->init(dev);
21.     ......
22. }
23.
24. if (!dev_valid_name(dev->name)) {
25.     ......
26. }
27.
28. dev->ifindex = dev_new_index();
29. if (dev->iflink == -1)
30.     dev->iflink = dev->ifindex;
31.
32. /* 检查是否有同名设备 */
33. head = dev_name_hash(dev->name);
34. hlist_for_each(p, head) {
35.     struct net_device *d
36.         = hlist_entry(p, struct net_device, name_hlist);
37.     if (!strncmp(d->name, dev->name, IFNAMSIZ)) {
        说明已经有一个同名同姓的设备已经在系统中了，
        那么你最好给你的设备起一个独一无二的名字
38.         ret = -EEXIST;
39.         ......
40.     }
41. }
42. ......
43. /*
44.  *  nil rebuild_header routine,
45.  *  that should be never called and used as just bug trap.
46.  */
47.
48. if (!dev->rebuild_header)
49.     dev->rebuild_header = default_rebuild_header;
   把设备放入 net_class 类中的目录下，这里面比较复杂而无趣，主要是 sysfs 管理驱动程序的例程，不管它了。
50. ret = netdev_register_sysfs(dev);
51. ......
    明确该设备已经注册过了，以后在进入此函数将会产生一个错误
52. dev->reg_state = NETREG_REGISTERED;
```

注意这里只是检查 index 和 name 的合法性，后面才把 dev 放入这两个 hash 表中。

```
53. /*
54.  *   把设备放入 dev_base 链表中，这里 dev_tail 也是一个全局变量
55.  */
56.
57. set_bit(__LINK_STATE_PRESENT, &dev->state);
58.
59. dev->next = NULL;
60. *dev_tail = dev;
61. dev_tail = &dev->next;
62. 下面才是把设备分别放入 name 表和 index 表
63. hlist_add_head(&dev->name_hlist, head);
64. hlist_add_head(&dev->index_hlist, dev_index_hash(dev->ifindex));
65.
66.
67. /* Notify protocols, that a new device appeared. */
68. raw_notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
69.
70. ret = 0;
71.
72. out:
73. return ret;
74.
75. }
```

代码段 2-34 register_netdevice 函数

下面这个结构就是刚才一直说的 net_device，它显得非常庞大，这其实是 Linux 内核目前不适合用作高性能由器操作系统的地方。

```
1.  /*
2.   *  Actually, this whole structure is a big mistake.  It mixes I/O
3.   *  data with strictly "high-level" data, and it has to know about
4.   *  almost every data structure used in the INET module.
5.   *
6.   *  FIXME: cleanup struct net_device such that network protocol info
7.   *  moves out.
8.   */
9.
10. struct net_device
11. {
12.
13. /*
14.  * This is the first field of the "visible" part of this structure
15.  * It is the name the interface.
16.  */
17. char            name[IFNAMSIZ];
18.
19. /*
20.  *  I/O specific fields
21.  *  FIXME: Merge these and struct ifmap into one
22.  */
23. ulong       mem_end;    /* shared mem end   */
24. ulong       mem_start;  /* shared mem start */
     x86 处理器主要使用 IO 地址空间，而其他架构得处理器使用内存映射 IO，IRQ 是设备中断号，x86 处理器也用
     它
25. ulong       base_addr;  /* device I/O address   */
26. uint        irq;        /* device IRQ number    */
27.
28. /*
29.  *  Some hardware also needs these fields, but they are not
30.  *  part of the usual set specified in Space.c.
31.  */
32.
33. uchar       if_port;    /* Selectable AUI, TP,..*/
34. uchar       dma;        /* DMA channel      */
35.
36. ulong       state;
```

```
37.
38. struct net_device   *next;
39.
40. /* 设备初始化函数，只被调用一次*/
41. int         (*init)(struct net_device *dev);
42.
43. /* ------- Fields preinitialized in Space.c finish here ------- */
44.
45. struct net_device   *next_sched;
46.
47. /* Interface index. Unique device identifier    */
48. int         ifindex;
49. int         iflink;
50.
51. struct net_device_stats* (*get_stats)(struct net_device *dev);
52.
53. /* 以下是和协议特定的数据 */
54. void            *ip_ptr;    /* IPv4 specific data   */
55. void                *ip6_ptr;      /* IPv6 specific data */
56. void            *ax25_ptr;  /* AX.25 specific data */
    还有一些网络协议的 ptr 指针我没有列出来
57. struct list_head    poll_list;  /* Link to poll list    */
58. int         quota;
59. int         weight;
```

每个网络设备对应一个此队列（也可以没有，例如 lo）。它将网络层与设备驱动区分开来。数据包在网络层处理完之后，可能会缓冲到此队列中。这里也只是可能会缓冲，因为数据包被送到此队列后，如果条件允许，立刻就被发送出去。

此队列是 Linux 内核中实现 QoS 的关键。不同的 QoS 策略采用不同的方式对此队列中的数据包进行处理，默认的方式是 FIFO。如果底层 driver 处理速度跟不上发送数据包的速度，那么当队列满了以后，后续的数据包就会被丢弃。

```
60. struct Qdisc        *qdisc;
61. struct Qdisc        *qdisc_sleeping;
62. struct Qdisc        *qdisc_list;
63. struct Qdisc        *qdisc_ingress;
64. unsigned long       tx_queue_len;   /* 每个队列中允许存放的最多报文数量 */
65.
66. /*
67.  * This marks the end of the "visible" part of the structure. All
68.  * fields hereafter are internal to the system, and may change at
69.  * will (read: may be cleaned up at will).
70.  */
71.
72. ushort      flags;  /* 接口标志，与 BSD 风格保持一致 */
73. ushort      gflags;
74. ushort  priv_flags; /* 和 'flags'一样，但是对用户空间不可见. */
75. ushort  unused_alignment_fixer; /* Because we need priv_flags,
76.                                          * and we want to be 32-bit aligned.
77.                                          */
78.
79. uint        mtu;    /* interface MTU value      */
80. ushort      type;   /* 接口的硬件类型，只在组 ARP 包的时候有用，下表是常见的类型  */
```

宏	值	说明
ARPHRD_ETHER	1	普通以太网（10Mbps）
ARPHRD_IEEE802	6	IEEE 802.2 以太网/TR/TB

```
81.
82. ushort      hard_header_len;    /* hardware hdr length  */
    指向设备特定数据，其实这段数据就紧跟在 net_device{}结构后面
83. void            *priv;
84.
85. struct net_device   *master; /* 指向一个组里的主（Master）设备，比如本书后面提到的聚合链路组，
    其中必有一个主链路，这里的 master 指针就指向这个主链路的设备 */
86.
87. /* 接口地址信息. */
88. uchar       broadcast[MAX_ADDR_LEN];    /* hw bcast add */
```

```
89. uchar       dev_addr[MAX_ADDR_LEN]; /* 这就是我们常说的 MAC 地址 */
90. uchar       addr_len;   /* 硬件地址长度，对于以太网设备，必然是 6 */
91.
92. int         promiscuity;
93.
94. int         watchdog_timeo;   /* 链路状态扫描时间间隔，后面的章节会提到 */
95. struct timer_list   watchdog_timer; /* 扫描链路状态的定时器函数结构 */
96.
97. /* 此变量将会加入设备名字 hash 链，就是 dev_name_head 变量*/
98. struct hlist_node   name_hlist;
99. /*此变量将会加入设备名字 hash 链，就是 dev_index_head 变量*/
100.    struct hlist_node   index_hlist;
101.
102.    /* register/unregister 状态机 */
103.    enum { NETREG_UNINITIALIZED=0,
104.           NETREG_REGISTERING,   /* called register_netdevice */
105.           NETREG_REGISTERED,    /* completed register todo */
106.           NETREG_UNREGISTERING, /* called unregister_netdevice */
107.           NETREG_UNREGISTERED,  /* completed unregister todo */
108.           NETREG_RELEASED,      /* called free_netdev */
109.    } reg_state;
110.
111.    /* 关于网络设备的一些特征，常见的如下表 */
112.    int         features;
```

宏定义	值	说明
NETIF_F_SG	1	Scatter/gather IO
NETIF_F_IP_CSUM	2	Can checksum only TCP/UDP over IPv4
NETIF_F_NO_CSUM	4	不需要做 checksum. 比如 loopack 设备
NETIF_F_HW_CSUM	8	可以对所有报文做 checksum
NETIF_F_HW_VLAN_TX	128	通过硬件发送 VLAN 报文，这样可以加速发送
NETIF_F_HW_VLAN_RX	256	通过硬件接收 VLAN 报文，加速接收
NETIF_F_VLAN_CHALLENGED	1024	设备不支持 VLAN 报文，报文可能会被认为是错包而被设备丢弃

```
113.    /* 下面是接口设备特定的内核函数 */
114.        int         (*open)(struct net_device *dev);
115.        int         (*stop)(struct net_device *dev);
116.        int         (*hard_start_xmit) (struct sk_buff *skb,
117.                                struct net_device *dev);
118.    #define HAVE_NETDEV_POLL
119.        int         (*poll) (struct net_device *dev, int *quota);
120.        int         (*hard_header) (struct sk_buff *skb,
121.                            struct net_device *dev,
122.                            unsigned short type,
123.                            void *daddr,
124.                            void *saddr,
125.                            unsigned len);
126.        int         (*rebuild_header)(struct sk_buff *skb);
127.    #define HAVE_SET_MAC_ADDR
128.        int         (*set_mac_address)(struct net_device *dev,
129.                                void *addr);
130.    #define HAVE_PRIVATE_IOCTL
131.        int         (*do_ioctl)(struct net_device *dev,
132.                            struct ifreq *ifr, int cmd);
133.    #define HAVE_SET_CONFIG
134.        int         (*set_config)(struct net_device *dev,
135.                            struct ifmap *map);
136.    #define HAVE_HEADER_CACHE
137.        int         (*hard_header_cache)(struct neighbour *neigh,
138.                                struct hh_cache *hh);
139.        void            (*header_cache_update)(struct hh_cache *hh,
140.                                struct net_device *dev,
141.                                unsigned char *  haddr);
142.    #define HAVE_CHANGE_MTU
143.        int         (*change_mtu)(struct net_device *dev, int new_mtu);
144.
```

```
145.
146.        int         (*hard_header_parse)(struct sk_buff *skb,
147.                                  unsigned char *haddr);
148.        int         (*neigh_setup)(struct net_device *dev, struct neigh_parms *);
149.        int         (*accept_fastpath)(struct net_device *, struct dst_entry*);
150.    #ifdef CONFIG_NETPOLL_RX
151.        int         netpoll_rx;
152.    #endif
153.    #ifdef CONFIG_NET_POLL_CONTROLLER
154.        void                 (*poll_controller)(struct net_device *dev);
155.    #endif
156.
157.        /* 桥端口的属性 */
158.        struct net_bridge_port  *br_port;
159.
160.    #ifdef CONFIG_NET_DIVERT
161.        /* this will get initialized at each interface type init routine */
162.        struct divert_blk   *divert;
163.    #endif /* CONFIG_NET_DIVERT */
164.
165.        /* class/net/name entry */
166.        struct class_device class_dev;
167.    };
```

代码段 2-35　net_device 结构

- 有一类设备，在协议栈里非常特殊，在大多数的 TCP/IP 实现里，都实现了这样一个设备，不过，它不是真正的设备，而只是一个虚拟的，用来做调试的接口。它就是 loopback 设备。在协议栈初始化的时候，这个接口必然要创建，而且一直存在。现在，我们来看看这个接口是如何初始化的，刚才提到了，它在被装入的时候，主动调用了 register_netdev，而不是由 PCI 模块回调它，原因很简单，没有实际设备的 id 与 loopback 设备相符，PCI 自然不会调用它，所以注册的事情就必须自己动手，丰衣足食啦。与 2.4 内核不同的是，在 2.6 内核中，loopback 接口设备的初始化被移到 net_olddevs_init 中。
- 当网卡是以模块动态加载方式初始化的时候，netdev_boot_base 返回 0。因为 net_olddevs_init 在各模块加载之前执行
- 当网卡以嵌入内核代码方式初始化的时候，netdev_boot_base 回返回 1。因为嵌入模块已经被初始化了，所以在 net_olddevs_init 函数执行的时候可以扫描到设备的存在。

下面是 loopback 设备的定义：

```
1.  struct net_device loopback_dev = {
2.      .name           = "lo",
3.      .mtu            = (16 * 1024) + 20 + 20 + 12,
4.      .hard_start_xmit    = loopback_xmit,
5.      .hard_header        = eth_header,
6.      .hard_header_cache  = eth_header_cache,
7.      .header_cache_update    = eth_header_cache_update,
8.      .hard_header_len    = ETH_HLEN, /* 14   */
9.      .addr_len       = ETH_ALEN, /* 6    */
10.     .tx_queue_len       = 0,
11.     .type           = ARPHRD_LOOPBACK,  /* 0x0001*/
12.     .rebuild_header     = eth_rebuild_header,
13.     .flags          = IFF_LOOPBACK,
14.     .features       = NETIF_F_SG | NETIF_F_FRAGLIST
15.                   | NETIF_F_NO_CSUM | NETIF_F_HIGHDMA
16.                   | NETIF_F_LLTX,
17. };
```

代码段 2-36　loopback_dev 结构

net_device{}结构中有 2 个非常重要的指针：ip_ptr 和 priv。ip_ptr 指向的是更高层次的数据结构，而 priv 指向的是设备底层的私有数据。因为在 Linux 内核维护人员来看，大部分网络设备具有的共性，和与物理硬件无关的数据可以归纳为一个数据结构，这是设备抽象层维护的结构，而硬件自己特有的数据比

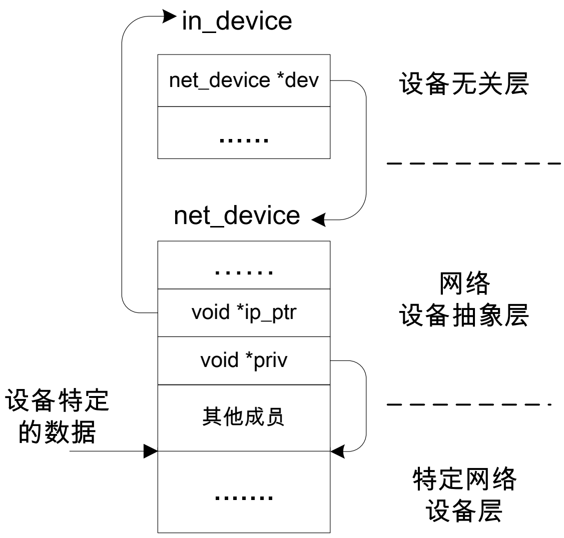
如寄存器和硬件缓冲区由驱动开发人员维护，但也不确定这些数据的大小和类型，于是使用一个 void 指针来指向。要注意的是 net_device 和设备私有数据是放在连续的空间的，请读者自行去参考 alloc_netdev 函数，这减少了申请内存的次数，也不会导致 2 次释放内存。

而且 net_device{}这个结构也不必被上层的协议栈操作，比如设备可能有 IP 地址，也可能没有 IP 地址，更重要的是，该设备也不应该只和 IP 协议打叫道，可能还有 PPP 协议、SLIP、X25 协议，所以应该再找一个跟此结构相关但却和协议相关的“代理结构”来记录这些信息。由于并不肯定是和 IP 协议栈交互，那么指向这个“代理结构”的指针类型还是 void 类型。当然按照命名规矩，在 IP 协议栈的框架内，提出了一个 in_device{}的结构，in 就是 ip network 的意思啦。虽然目前 net_device{}列出了 ip_ptr 和其他网络的指针，但我敢预言将来这个几个指针会用“联合”来代替，除非该设备不仅工作在 IP 网络上还工作在其他类型的网络上。

```
1.  struct in_device
2.  {
3.      struct net_device       *dev;
4.      atomic_t        refcnt;
5.      int         dead;
    维护设备的地址列表，对的，是列表，一个设备可以有多个地址，对于普通 PC 机来说似乎比较少见，但在服务器上比较常用。我们在后面的章节中会介绍它的
6.      struct in_ifaddr    *ifa_list;  /* IP ifaddr chain      */
    下面 m 开头这几个成员都是和组播有关系，我删去了好几个
7.      struct ip_mc_list   *mc_list;   /* IP multicast filter chain    */
8.      struct ip_mc_list   *mc_tomb;
9.      unsigned long       mr_v1_seen;
10.     ......
11.     维护 arp 协议互操作的参数
12.     struct neigh_parms  *arp_parms;
13.     struct ipv4_devconf cnf;
14. };
```

代码段 2-37 in_device 结构

这个结构和 net_device 的关系如下图，这个关系在 inetdev_init 函数中确定：



图表 2-26net_device 和 in_device、设备特定数据之间的关系

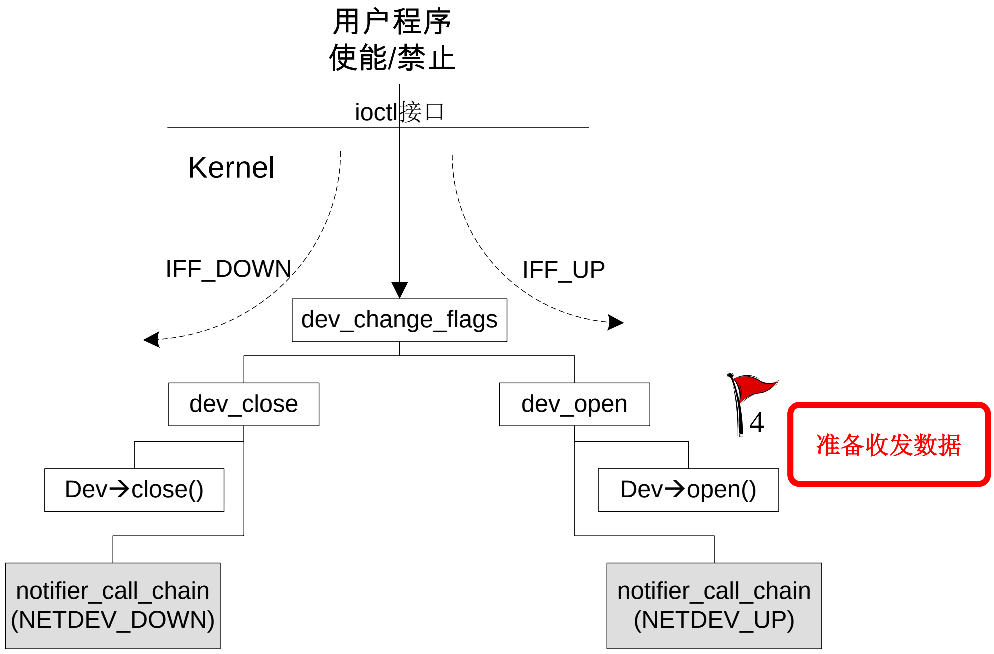
- 设备无关层采用 in_device{}数据结构保存 IP 地址和邻居信息——虽然是间接的
- 网络抽象层采用 net_device{}数据结构保存设备的名字、编号、地址等共性
- 设备特定层的数据则有设备驱动开发人员自己定义，一般有硬件发送、接收缓冲区、芯片寄存器的信息等等。这片内存区一般是紧跟在 net_device{}后面，由驱动程序在创建 net_device{}的时候顺带把这块内存也创建了。当然还是用 priv 指针指向，以方便访问。

设备已经被注册了，那么是否就可以工作了呢？不是的，还得靠用户把这些网卡激活，当然，一般

都有脚本在系统初始化的时候干这样的活。网卡被激活的时候，它要完成几个非常重用的事情：

1. 挂接中断处理函数（ISR），如果不能为驱动程序申请到中断，那说明要么网卡没插好，要么和其他设备发生了冲突，结果就是设备根本不能用。
2. 创建驱动程序内部接收环和发送缓冲区，网卡一般都要“环”的方式来存放报文。
3. 挂接接口状态扫描定时器，以 poll 的方式轮询接口是否真正 up 或 down。
4. 进一步打开设备特点寄存器，使其可以开始收发报文了

我将这个过程画在下面这幅图中。



看到这几幅图，读者一定都看到了那几个红色小红旗及方框，其中写明了此时代码要完成的基本任务。现在可以对驱动程序的初始化做一个基本的概括，即 Linux2.6 下网络驱动程序的初始化分为 4 个基本步骤：

第1步.　系统把驱动程序装入内存
第2步.　PCI 为设备选择正确的驱动程序，并分配相应内存数据结构
第3步.　指定驱动程序如何处理报文格式
第4步.　用户打开设备使其可以真正工作起来

关于底层驱动的架构已经基本介绍完了，在这里我还得提醒读者，在本书中，设备就是接口，接口就是设备，它们在一般情况下是同等意义。也许有读者提出一块网卡可能有 2 个接口，那么这算多少设备呢？可以这样解答：设备驱动程序可以只有一份，但是你创建的 net_device{}结构必须要 2 个，也就是说，Linux 内核本来不是用来做路由器的，现在你要赶鸭子上架硬是要实现路由器级别的 Linux，那么你得忍受内核中一些编码风格——那个 net_device 实际可以改名为 net_interface 等。所以，本书中“接口”和“设备”是换着使用，它们的含义基本上是接近“接口”概念的。

第3章　配置系统

3.1　配置过程分析

在本章，我们先介绍配置普通设备的 IP 地址的内部过程，接着再转到 loopback 接口的配置过程，这两个过程有相似之处，所以一起解说。然后再转入 FIB 系统，讲解我们最感兴趣的路由系统，并用图例演示路由表的变化。最后介绍接口状态的变化，这对于驱动程序开发人员来说也是比较重要的。

3.1.1　配置是如何下达到内核的？

我们假设在安装我们的 Linux 系统时，没有配置 IP 地址，也没有挂上网线，完完全全是一台“裸机”，这样方便我们跟踪系统到底做了什么。

安装完系统之后，我们可以使用 ifconfig 命令，查看设备信息。我想读者们应该都知道这么一个命令吧。在 Windows 上相应的操作是 ipconfig。带上”-a”参数表示要查看详细的配置，包括 loopbcack 设备。

#ifconfig –a

eth0　　Link encap:Ethernet　HWaddr 00:80:C8:EB:2A:39
BROADCAST MULTICAST　MTU:1500　Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B)　TX bytes:0 (0.0 b)
Interrupt:5

lo　　Link encap:Local Loopback
inet addr:127.0.0.1　Mask:255.0.0.0
UP LOOPBACK RUNNING　MTU:16436　Metric:1
RX packets:6 errors:0 dropped:0 overruns:0 frame:0
TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 b)　TX bytes:0 (0.0 b)

不同的机器和网卡会显示一些不同。在我的另外一台机器，其 eth0 的 Interrupt 等于 3。

现在运行**#strace ifconfig eth0 192.168.18.2 netmask 255.255.255.0**，配置好 ip 地址和网络掩码。为什么要运行 strace？在大部分系统上是没有 ifconfig 的源代码的，那么为了查看 ifconfig 内部完成了什么操作时，可以用 strace 命令查看。它收集应用程序执行的系统调用，甚至参数都能记录下来。通过对这个命令的输出进行整理，我们可以把 ifconfig 内部调用的系统接口整理如下：

```
1.  main(int argc, char **argv )
1.  {
2.      struct sockaddr sa;
3.      struct sockaddr_in sin;
4.      char host[128];
5.      struct aftype *ap;
6.      struct hwtype *hw;
7.      struct ifreq ifr;
8.      char **spp;
9.      int fd;
10.
11.     fd = socket(PF_INET, SOCK_DGRAM, IPPROTO_IP);
12.     ifr.ifr_name = “eth0”;
13.    ap = inet_aftype =
14.    {
15.        "inet", NULL, /*"DARPA Internet", */ AF_INET, sizeof(unsigned long),
16.        INET_print, INET_sprint, INET_input, INET_reserror,
17.        NULL /*INET_rprint */ , NULL /*INET_rinput */ ,
18.        INET_getnetmask,
19.        -1, /* 这个值会被赋成 fd，即刚才打开的 socket */
20.        NULL
21.    };
```

```
22.    host = "192.168.18.2";
23.    ap->input(0, host, &sa);/* 在此 sa→sa_family=AF_INET，af→sa_data 已经被设置成
       192.168.1.1 */
24.    memcpy((char *) &ifr.ifr_addr, (char *) &sa, sizeof(struct sockaddr));
25.    ioctl(fd, SIOCSIFADDR, &ifr);
26.    ioctl(skfd, SIOCGIFFLAGS, &ifr);
27.    ioctl(skfd, SIOCSIFFLAGS, &ifr);
28.    /* 开始第二次遍历命令行 */
29.    host = "255.255.255.0";
30.    ioctl(skfd, SIOCSIFNETMASK, ifr);
31.
32.  }
```

代码段 3-1配置 ip 地址的用户层代码

以上就是 ifconfig 这个命令实际内部"伪"代码，为了减轻复杂度，我们不必把整个代码列出来。从上面的代码中可以看出 ifconfig 实际调用了 2 个系统函数：socket 和 ioctl。4 表示为 socket 打开的文件描述符，姑且认为这是应用程序到达系统内核的一个钥匙吧，后面会介绍一些相关知识。Socket 的参数以后也会详细介绍。然后是 ioctl 这个系统调用。在 Linux 相关的项目中，ioctl 是用户层与内核或设备驱动程序进行配置的一个有效手段。在这里我们要配置系统的地址，则必定要通过 ioctl。在 ifconfig 的程序中依次调用 6 个 ioctl，完成了对系统地址的配置。

下面我们就对出现的系统调用进行分析，首先是 socket。

3.1.2 socket 系统调用

从上图可以看到一个已安装的 Linux 操作系统究竟支持几种文件系统类型由文件系统的注册链表决定。

在 BSD socket 层内使用 msghdr{ }结构保存数据；在 INET socket 层以下都使用 sk_buff{ }数据结构保存数据。

这一部分处理 BSD socket 相关操作，每个 socket 在内核中以 struct socket 结构体现

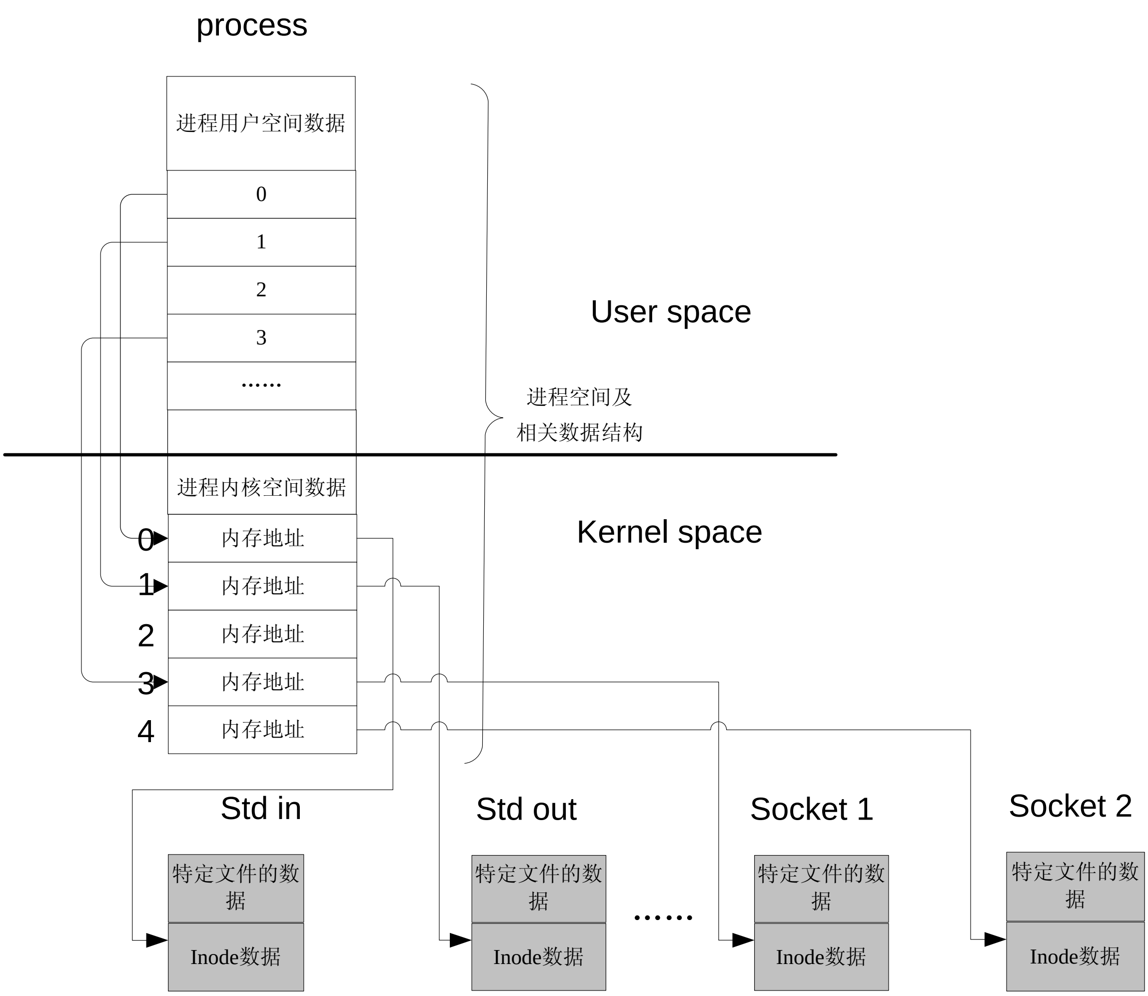
```
struct socket {
        struct proto_ops      *ops;
        struct file        *file;
        struct sock           *sk;
        wait_queue_head_t wait;
        short              type;
        ……
};
```

应用层中的操作对象是 socket 文件描述符，通过文件系统定义的通用接口，使用系统调用从用户空间切换到内核空间，控制 socket 文件描述符对应的就是对 BSD socket 的操作，从而进入到 BSD socket 层的操作。在 BSD socket 层中，操作对象是 socket{ }结构。每一个这样的结构对应的是一个网络连接。通过网络地址族的不同来区分不同的操作方法，判断是否应该进入到 INET socket 层，这一层的数据存放在 msghdr{ }结构中。在 INET socket 层中，根据建立连接的类型，分成面向连接的和面向无连接两种类型，这是区分 TCP 和 UDP 协议的主要原则。这一层的操作对象主要是 sock{ }类型的数据，而数据存放在 sk_buff{ }结构中。从 INET socket 层到 IP 层，主要是路由过程，发送时根据发送的目标地址确定需要使用的网络设备接口和下一需要传送的机器地址。

在内核中与 socket 对应的系统调用是 sys_soceket，所谓的创建套接口，就是在 sockfs 这个文件系统中创建一个节点，从 Linux/Unix 的角度来看，该节点是一个文件，不过这个文件具有非普通文件的属性，于是起了一个独特的名字——socket。由于 sockfs 文件系统是系统初始化时就保存在全局指针 sock_mnt 中的，所以申请一个 inode 的过程便以 sock_mnt 为参数。

从进程角度看，一个套接口就是一个特殊的已打开文件。现在将 socket 结构的宿主 inode 与文件系统挂上钩，就是分配文件号以及 file 结构在目录数中分配一个 dentry 结构。指向 inode 使 file->f_dentry 指向 inode 建立目录项和索引节点（即套接口的节点名）

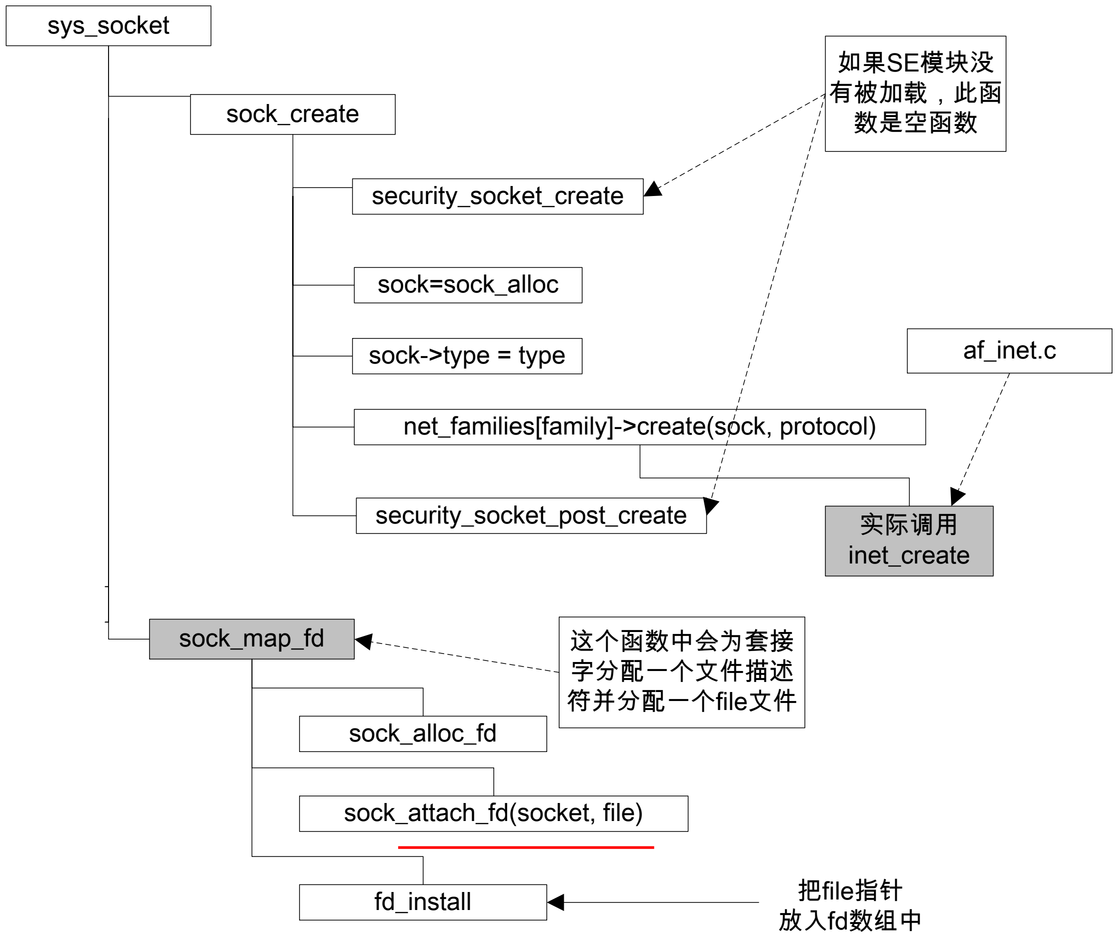
现在我们来看看每个进程是如何对待其打开的文件及 socket 的：



图表 3-1FD 的意义

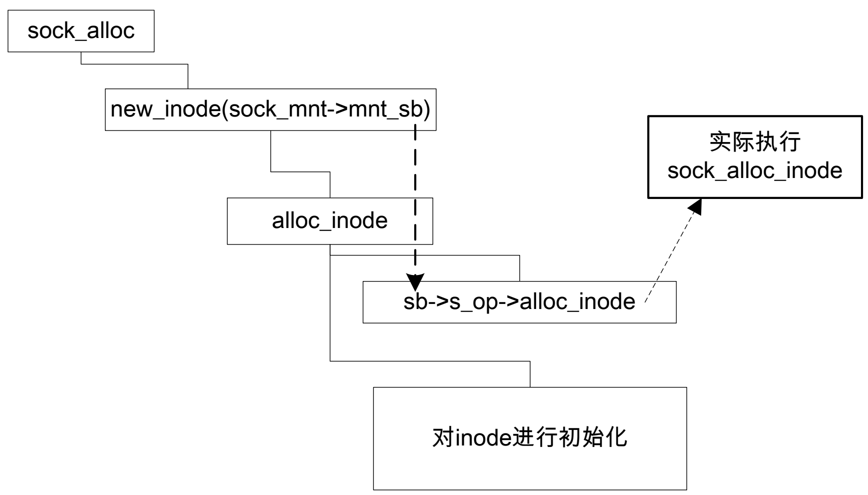
从上图中可以看到，标准输入接口（std in）的文件描述符占据了进程的用户空间的文件描述符数组的第一个单元，这是系统内定的。通过内核的组织，其在内核中的描述符数组的第一个单元存放了指向其 inode 的地址，以此类推，标准输出接口（std out）占据第二个单元，错误输出接口（std err）占据第三个单元（上图没有画出）。而用户自己打开的文件或 socket，则依次排列到系统已经内定的错误输出接口的 fd 单元后面，比如，如果某进程第一次打开了一个常规文件，它的 fd 必定是 3，当再打开别的文件时，fd 就是 4，这里 3 和 4 就是用户空间中的 fd 数组下标。同样的道理，socket 也可以看作是一个文件。每次调用 socket 返回的 fd 值也是指用户空间的 fd 数组下标。

现在我们来看看 socket 函数本身，经过 glibc 库对其封装，它将通过 int 0x80 产生一个软件中断（注意不是软中断），由内核导向执行 sys_socket，基本上参数会原封不动地传入内核，它们分别是(1) int family, (2) int type, (3) int protocol。其调用树如下图：



图表 3-2sys_socket 的函数调用树

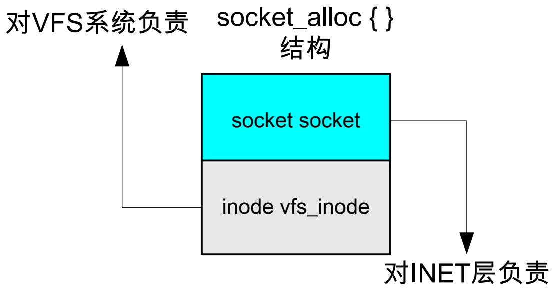
由于不关心 security 方面的代码，我们就不研究 security_socket_create 函数了。直接对 sock_alloc 进行分解，struct socket{}结构就是它创建的，它的调用树如下：



图表 3-3sock_alloc 函数调用树

我们的 mnt_sb→s_op→alloc_inode 就是前文中静态全局变量 super_operations sockfs_ops 定义的 sock_alloc_inode 函数。它仅仅是用 kmem_cache_alloc 创建一个 socket_alloc{}类型的 inode，然

后返回给 alloc_inode 进行初始化。所以纵观 sock_alloc( )的作用就是分配及初始化属于网络类型（应该是 socket_alloc 类型）的 inode。inode 结构中的大部分字段只是对真正的文件系统重要，但是，只有一部分由 socket 使用。socket_alloc 类型的 inode 结构如下：



图表 3-4 soket_alloc 结构

此图中的 socket socket 部分就是"FD 的意义"一图中那个"特定文件的数据"部分。那么 socket{ } 结构就表示这是一个跟网络有关的文件描述符，是 INET 层与应用层打开的文件描述一一对应的实体，每一次调用 socket 函数都会在 INET 中保存这么一个实体。

在常规文件系统中常用的 open 系统调用不能用来创建一个 socket，sockets 只能用 socket API 创建。一旦 socket 被创建，IO 操作就和普通文件或设备一样了。sock_alloc 实际上创建了 socket 结构并从 socket inode slab cache 中创建 inode 结构。SOCKET_I 和 SOCK_INODE 可以相互转换 inode 和 socket。内核通过 fd 找到内存中对应的 inode，然后再通过 VFS 系统查找到对应的内核模块，这样用户空间和内核协议栈就搭上了关系。一旦 inode 被创建，socket 层就可以映射 IO 系统调用了。一个 file_operations 结构，socket_file_ops 被创建并初始化，其中所有的 IO 系统调用都有对应的 socket 对应操作

（注：open 调用返回一个 ENXIO 错误）

```
1.  /*
2.   Socket 类型的文件系统和其他文件系统一样有一个特别的操作集合， 但是有相当一部分操作是经过直接
    调用系统接口而不是通过这个操作集合来完成。
3.   */
4.
5.  static struct file_operations socket_file_ops = {
6.      .owner =    THIS_MODULE,
7.      .llseek =   no_llseek,
8.      ......
9.      .poll =     sock_poll,
10.     .unlocked_ioctl = sock_ioctl,
11.     .open =     sock_no_open,   /* 这个 open 操作明显是 socket 系统不支持的 */
12.     .release =  sock_close,
13.     ......
14.     .readv =    sock_readv,
15.     .writev =   sock_writev,
16.     .......
17. };
```

代码段 3-2 socket_file_ops 结构定义

到此为止，我们可以作出结论，VFS 为了使 socket 系统工作——或者说 socket 为了适应 VFS 系统框架，socket 提供了 2 个数据结构：super_operations——sockfs_ops 和 file_operations——socket_file_ops。前者是必须的，它创建了 VFS 必需的 inode，使 VFS 可以对其进行文件级别的管理；而后者是可选的，用户层可以使用标准文件系统的操作比如 write( )和 read( )对 socket 对象进

行操作，也可以采用系统提供的 send( )和 recv( )接口对 socket 对象进行处理，但二者都归一到网络内部实现代码中。

在 ifconfig 的代码中 socket 系统调用的第一个参数即 family 是 PF_INET，意味着 socket_create 中的 net_families[family]->create(sock, protocol)指的是 net_families[PF_INET]里的 inet_create 函数，它在“网络系统初始化”中提及。而第二个参数 type 被赋给 sock→type 成员，而后根据这个 type 获取 inetsw[]数组中相对应的成员。

我们再来看 inet_create 代码：

```
1.  /*
2.   * 创建一个 inet 套接字
3.   */
4.  static int inet_create(struct socket *sock, int protocol)
5.  {
6.  struct sock *sk;
7.  struct list_head *p;
8.  struct inet_protosw *answer;
9.  struct inet_sock *inet;
10. struct proto *answer_prot;
11. unsigned char answer_flags;
12. char answer_no_check;
13. int try_loading_module = 0;
14. int err;
15.
16. sock->state = SS_UNCONNECTED;
17.
18. /* Look for the requested type/protocol pair. */
19. answer = NULL;
20. lookup_protocol:
21. err = -ESOCKTNOSUPPORT;
22.
23. list_for_each_rcu(p, &inetsw[sock->type]) {
24.     answer = list_entry(p, struct inet_protosw, list);
25.
        由于 UDP→protocol 为 IPPROTO_UDP(17)，TCP→protocol 为 IPPROTO_TCP(6).
        if (protocol == answer->protocol) {
        目前只有只有 RAW IP→protocol 为 IP(0)，如果应用程序传入的值也为 IP(0)，那么会直接走
        到 36 行
26.         if (protocol != IPPROTO_IP)
27.             break;
28.     } else {
            在 ifconfig 应用程序中走这个分支
29.         if (IPPROTO_IP == protocol) {
30.             protocol = answer->protocol;
31.             break;
32.         }
            在 ping 这种应用（后面章节会介绍）中走这个分支，传入的 protocol 为 ICMP(1)，而且，
            如果你使用<SOCK_RAW, OSPF(89)>的组合时也会走这个分支，即使用 IP 层数据传输。
33.         if (IPPROTO_IP == answer->protocol)
34.             break;
35.     }
        在使用<SOCK_RAW, IP>的组合时会造成创建 socket 失败。
36.     err = -EPROTONOSUPPORT;
37.     answer = NULL;
38. }
39. err = -EPERM;
40. if (answer->capability > 0 && !capable(answer->capability))
41.     goto out_rcu_unlock;
        在 ifconfig 这个例子中，这个 answer 指向的是 UDP，而其 ops 指向 inet_dgram_ops，而 prot
        指向 udp_prot
42. sock->ops = answer->ops;
43. answer_prot = answer->prot;
44. answer_no_check = answer->no_check;
45. answer_flags = answer->flags;
```

这里是 Linux 与其它操作系统不同的一个地方，比如 BSD 或 VxWorks 允许<RAW，IP>的组合，但其目的和 Linux 的<DGRAM，IP>一样。

```
46.
47.
48. err = -ENOBUFS;
    一旦 inode 被创建，sock_alloc 会从这个 inode 中得到 socket 结构。然后初始化这个 socket 结
    构的某些字段。并且 fasync_list 被设置成 NULL。要记住，sockets 并不仅仅是为 TCP/IP 而建，
    保存在 socket 结构中的状态并不与 TCP 的相似，TCP 的会更加复杂。socket 状态实际上只反映了是
    否这是一个活动的连接。
49. sk = sk_alloc(PF_INET, GFP_KERNEL, answer_prot, 1);
50.
51. err = 0;
52. sk->sk_no_check = answer_no_check;
53. if (INET_PROTOSW_REUSE & answer_flags)
54.     sk->sk_reuse = 1;
55.
56. inet = inet_sk(sk);
57. inet->is_icsk = INET_PROTOSW_ICSK & answer_flags;
58.
59. if (SOCK_RAW == sock->type) {
        专门给 RAW IP 设置了一个端口号。我们会在下面看到这个端口号有什么用。此时 num 等于 1
        （IPPROTO_ICMP）
60.     inet->num = protocol;
61.     if (IPPROTO_RAW == protocol)
62.         inet->hdrincl = 1;
63. }
64.
65. if (ipv4_config.no_pmtu_disc)
66.     inet->pmtudisc = IP_PMTUDISC_DONT;
67. else
68.     inet->pmtudisc = IP_PMTUDISC_WANT;
69.
70. inet->id = 0;
    为每个打开的 socket 设置其等待队列，我们会在“select 实现”的章节中再次看到它
71. sock_init_data(sock, sk);
72.
73. sk->sk_destruct   = inet_sock_destruct;
74. sk->sk_family     = PF_INET;
75. sk->sk_protocol   = protocol;
76. sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
77.
78. if (inet->num) {
79.      在此我们见识到 num 和 sport 实际就是同一个值，只是在使用主机字节序的情况下使用 num，而
    在网络字节序的情况下使用 sport
80.     /* 我们假设任何协议都允许用户在创建 socket 的时候指派一个号码 */
81.     inet->sport = htons(inet->num);
82.     /* 把这个 sock 加到协议 hash 链中*/
83.     sk->sk_prot->hash(sk);
84. }
    udp_prot→init 实际上等于 NULL，所以对 ifconfig 这个应用及所有基于 UDP 应用程序而言，下面
    的判断肯定不会是真，而对于 TCP 和 IP 应用则必须执行相应的 init 函数，后面介绍 TCP 时会再次碰
    到下面两行
85. if (sk->sk_prot->init) {
86. err = sk->sk_prot->init(sk);
87.
88. }
89. out:
90. return err;
91. out_rcu_unlock:
92.
93. goto out;
94. }
```

```
static void raw_v4_hash(struct sock *sk)
{
    struct hlist_head *head
        = &raw_v4_htable[inet_sk(sk)->num &
                         (RAWV4_HTABLE_SIZE - 1)];
    sk_add_node(sk, head);
}
```

代码段 3-3 inet_create 函数

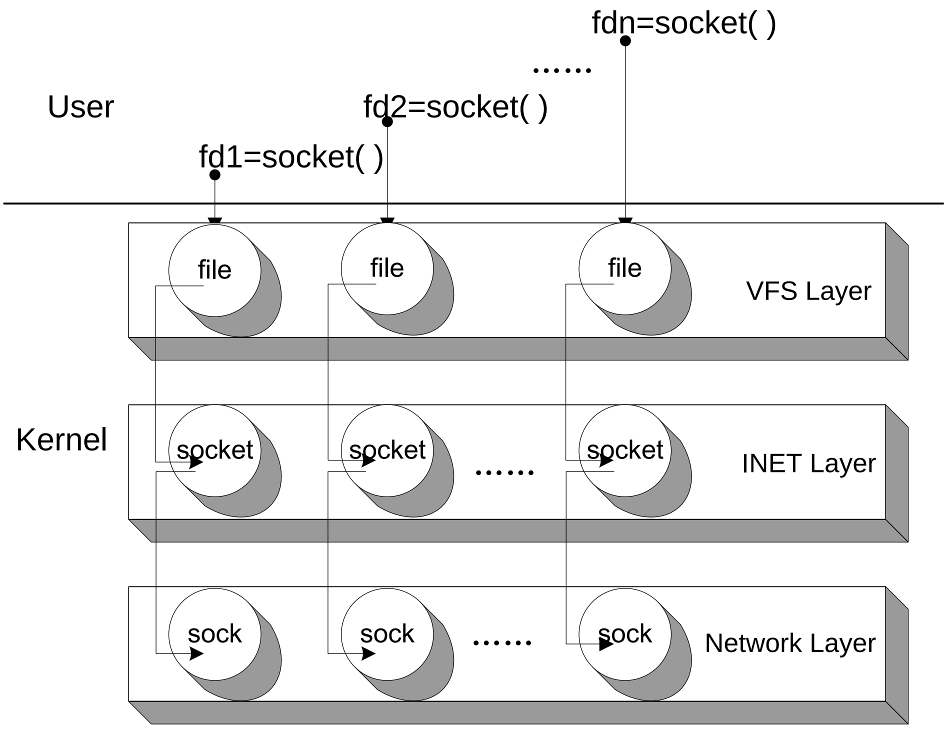
在这段代码中又发现一个极易引起混乱的结构体名字——sock{ }，它和 socket{ }结构的名字只差了两个字母，它和 sock 确实有关系。为何？socket{}结构表示 INET 中的实体，而 sock{ }结

构就是网络层实体，sock 用来保存打开的连接的状态信息。它一般定义的变量叫 sk。等会就给你演示它们之间的关系。

socket 系统调用的第三个参数是 protocol，在这里它终于派上用场——赋给 sk->sk_protocol，不过，在 ifconfig 这个应用中，它没有起到作用。

backlog_rcv 在此根据协议分别是 tcp_v4_do_rcv，udp_queue_rcv_skb 或者是 raw_rcv_skb，在 ifconfig 的这个应用中，则是第二个。不过，由于目前只是配置，我就不在这里介绍它了，而是放在后面去分析这个函数。

根据对 sys_socket 的分析，我们可以推断出 file{ }、socket{ }、sock{ }之间的关系如下：

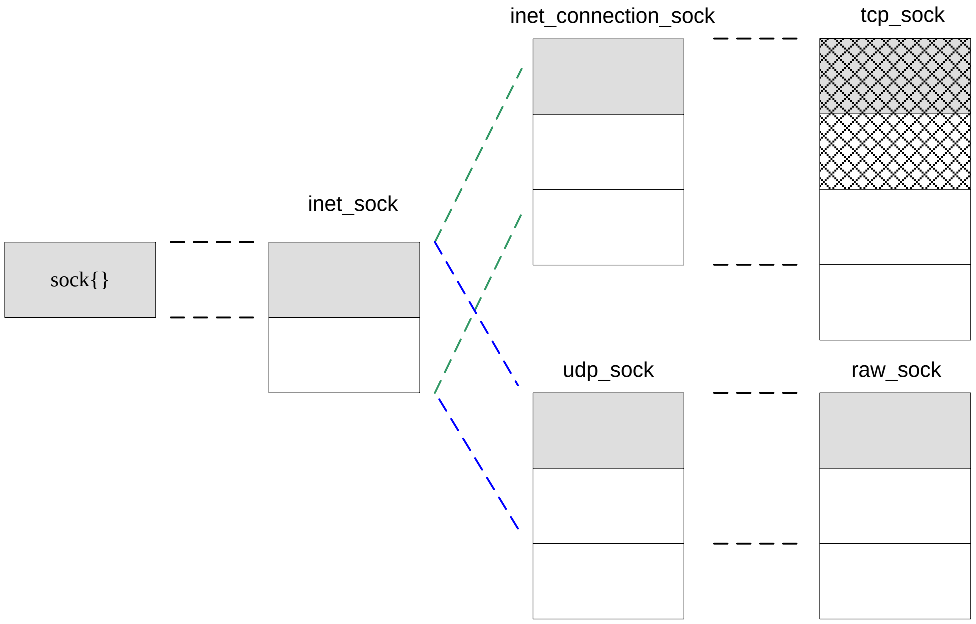


图表 3-5file、socket、sock 之间的关系

会通过 sock_map_fd 指向一个 file 指针。这个文件指针用来维护与该 socket 相关联的伪文件的状态。也就是说，不管你的应用程序是基于 TCP 的还是基于 IP 甚至是跟网络传输没有关系（例如 ifconfig），内核会在 INET 和 Network 层分别创建一个实体来对应你打开的那个文件描述符，而后你对该文件描述符的操作都是通过这两个数据结构的实体去完成。因为系统为每次 socket 系统调用都创建一对<file, socket，sock>三元组，所以每个应用程序打开的每一个文件描述符都不会出现互斥操作，也就避免了锁的开销。每次应用程序要访问内核模块时，都通过 fd 去操作，内核会在 fd 对应的数组单元内查找到相应的 socket，然后才能将后继的操作进行下去，比如我们马上要分析的 ioctl。

在 inet_create 函数中创建 sock{}的时候调用 sk_alloc 接口，它根据协议的类型来创建真正的"sock"结构，因为协议不同，实际创建的对象不同，虽然函数返回值都是 sock{}指针，但是实际的大小并不相同，比如，对于 raw 应用，我们应该创建的是 raw_sock，而对于 tcp 来说，应该创建的是 tcp_sock，由于在创建 sock{}时就已经根据协议类型为其准备了足够大的空间，所以当

进入到协议相关的代码部分，使用指针类型强制转型，将其转换为相应的数据结构。



图表 3-6 sock 结构在不同协议的数据块

INET socket 层支持包括 TCP/IP 协议在内的 internet 地址族。如前所述，这些协议是分层的，一个协议使用另一个协议的服务。Linux 的 TCP/IP 代码和数据结构反映了这一分层模型。BSD socket 层从已注册的 INET proto_ops 数据结构中调用 INET 层 socket 支持例程来为它执行工作。例如，一个地址族为 INET 的 BSD socket 建立请求，将用到下层的 INET socket 的建立函数。为了不把 BSD socket 与 TCP/IP 的特定信息搞混，INET socket 层使用它自己的数据结构 sock，它与 BSD socket 结构相连。这一关系意味着后来的 INET socket 调用能够很容易地重新找到 sock 结构。sock 结构的协议操作指针也在初始化时建立，它依赖与被请求的协议。如果请求的是 TCP，那么 sock 结构的协议操作指针将指向 TCP 连接所必需的 TCP 协议操作集。

上图中演示了各种协议下不同 sock{}的本质，先列出一些相应的强转类型的宏或函数，给读者打好基础：

表格 3-1 sk 宏的输入输出比较

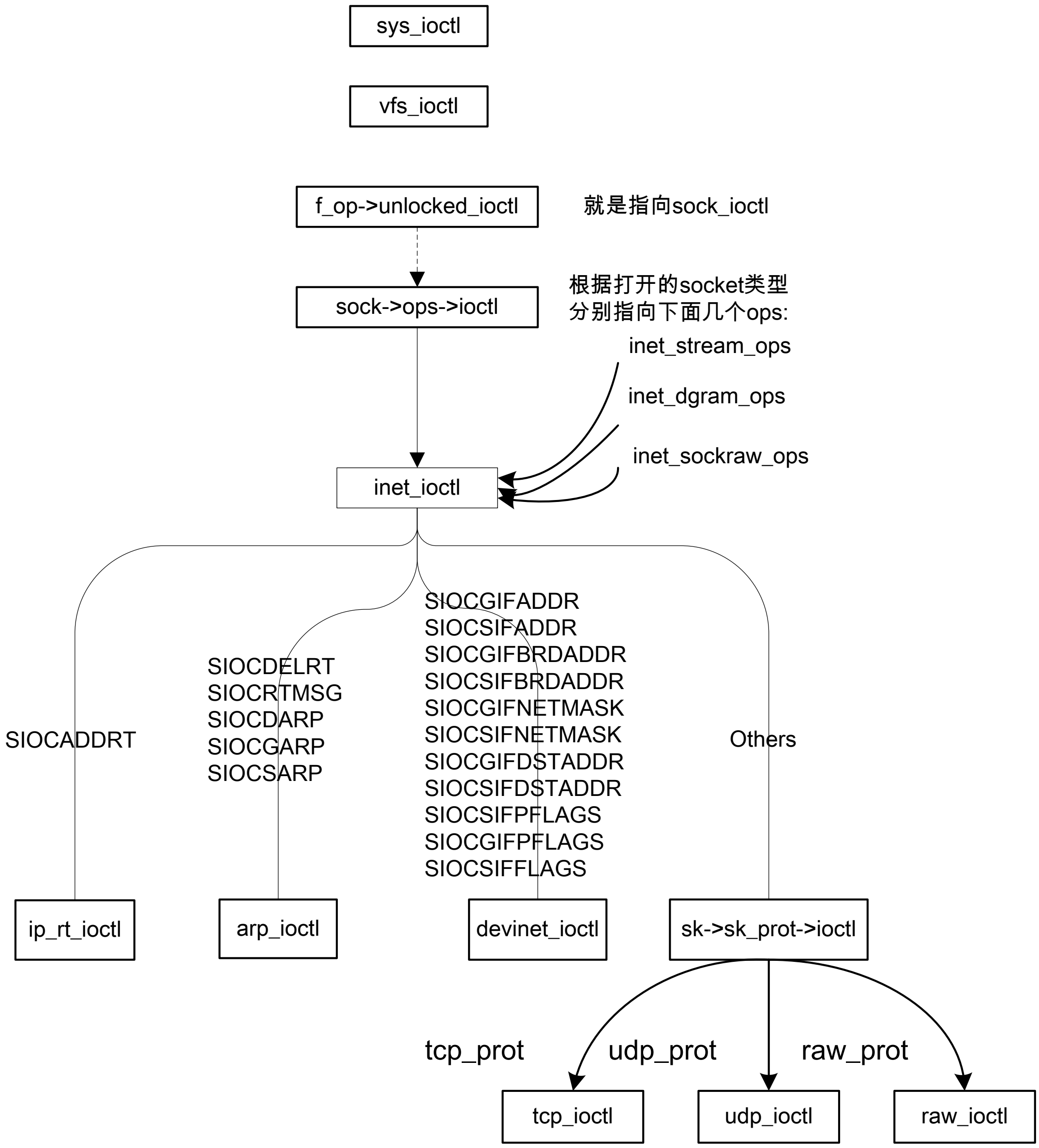
宏/函数	输入结构类型	输出结构类型
inet_sk	sock{}	inet_sock{}
udp_sk	sock{}	udp_sock{}
tcp_sk	sock{}	tcp_sock{}
raw_sk	sock{}	raw_sock{}

3.1.3 ioctl 代码的实现

ioctl 是标准库里的函数，实际上也就是说大部分操作系统都会支持这个系统调用。它是内核模块和应用程序交互配置的一种手段，而且，它是同步完成的，即它的代码必定处于某个进程的上下文中。读者们会问，还有什么交互手段不是这样的呢？答案是 netlink 接口。它是异步完成配置的工作的，比如读写路由表。先让我们目光集中在 ioctl 上。

3.1.3.1. 配置设备 IP 地址的内核处理过程

ioctl 函数在内核中对应的函数是 sys_ioctl，这个接口的操作属于 VFS 的，也就是说系统不区分用户要对哪种文件系统进行 ioctl，只有根据用户之前打开的文件类型来推断正确的调用路径。比如 socket 类型的文件系统，它的 ioctl 必定是由 socket_file_ops 结构定义的。内核中具体的执行路径如下：



图表 3-7ioctl 的内核实现

如果你要编写关于网络部分的 ioctl 应用程序代码，那么专门有一个数据结构来作为 ioctl 的参数给你，对于标准的内核，你必须使用该数据结构：

```
   Ifreq 表示对接口操作的“请求”
1.  struct ifreq
2.  {
3.      #define IFHWADDRLEN 6
4.      union
5.      {
      所有接口的 ioctl 必须有以 ifr_name 开始的参数定义
6.          char    ifrn_name[IFNAMSIZ];       /* if name, e.g. "en0" */
7.      } ifr_ifrn;
```

```
        下层函数根据 ioctl 的命令字来解析此联合的类型和值
8.       union {
9.           struct  sockaddr ifru_addr;
10.          struct  sockaddr ifru_dstaddr;
11.          struct  sockaddr ifru_broadaddr;
12.          struct  sockaddr ifru_netmask;
13.          struct sockaddr ifru_hwaddr;
14.          short   ifru_flags;
15.          int ifru_ivalue;
16.          int ifru_mtu;
17.          struct ifmap ifru_map;
18.          char    ifru_slave[IFNAMSIZ];  /* Just fits the size */
19.          char    ifru_newname[IFNAMSIZ];
20.          char    __user *    ifru_data;
21.          struct  if_settings ifru_settings;
22.     }
23.     ifr_ifru;
24. };
```

代码段 3-4 ifreq 结构

这个结构在内核中也有同样的定义，所以在 devinet_ioctl 函数就用 copy_from_user(&ifr, arg, sizeof(struct ifreq))这样一条语句搞定，arg 是 void *变量，实际在内部实现中已经变成了 unsigned long 变量，也就是指针变量。

Ifconfig 调用的 ioctl 码依次是 SIOCSIFADDR，SIOCSIFFLAGS，SIOCSIFNETMASK。

下面是 devinet_ioctl 函数的部分代码：

```
1.  int devinet_ioctl(unsigned int cmd, void *arg)
2.  {
3.      struct ifreq ifr;
4.      struct sockaddr_in sin_orig;
5.      struct sockaddr_in *sin = (struct sockaddr_in *)&ifr.ifr_addr;
6.      struct in_device *in_dev;
7.      struct in_ifaddr **ifap = NULL;
8.      struct in_ifaddr *ifa = NULL;
9.      struct net_device *dev;
10.     char *colon;
11.     int ret = -EFAULT;
12.     int tryaddrmatch = 0;
13.
14.     if (copy_from_user(&ifr, arg, sizeof(struct ifreq)))
15.         goto out;
16.     ifr.ifr_name[IFNAMSIZ - 1] = 0;
17.
18.     /* save original address for comparison */
19.     memcpy(&sin_orig, sin, sizeof(*sin));
20.
21.     colon = strchr(ifr.ifr_name, ':');
22.     if (colon)
23.         *colon = 0;
24.     ...... 以上省略了 GET 的操作，要注意
25.     ret = -ENODEV;
        根据用户指定的设备名参数来搜索设备，比如“lo”或者“eth0”
26.     if ((dev = __dev_get_by_name(ifr.ifr_name)) == NULL)
27.         goto done;
        如果用户指定“eth0：0”作为参数，就进入下面的复制
28.     if (colon)
29.         *colon = ':';
        在第一次对进行设备配置时，下面获取的 in_dev 是 NULL，因为此结构的创建是在后面进行的
30.     if ((in_dev = __in_dev_get(dev)) != NULL) {
31.         if (tryaddrmatch) {
        如果 in_dev 不是 NULL，说明曾经对该设备进行过配置，于是遍历 in_dev 上的 ifa 链表，根
        据名字和地址进行匹配，如果一致，说明我们要对同一个设备进行操作，在这里我们获得了 ifa。
32.             for (ifap = &in_dev->ifa_list; (ifa = *ifap) != NULL;
33.                  ifap = &ifa->ifa_next) {
```

```
34.                 if (!strcmp(ifr.ifr_name, ifa->ifa_label) &&
35.                     sin_orig.sin_addr.s_addr ==
36.                             ifa->ifa_address) {
37.                     break; /* found */
38.                 }
39.             }
40.         }
41.         /* we didn't get a match, maybe the application is
42.            4.3BSD-style and passed in junk so we fall back to
43.            comparing just the label */
44.         if (!ifa) {
45.             for (ifap = &in_dev->ifa_list; (ifa = *ifap) != NULL;
46.                  ifap = &ifa->ifa_next)
47.                 if (!strcmp(ifr.ifr_name, ifa->ifa_label))
48.                     break;
49.         }
50.     }
51.
52.     ret = -EADDRNOTAVAIL;
53.     if (!ifa && cmd != SIOCSIFADDR && cmd != SIOCSIFFLAGS)
54.         goto done;
55.
56.     switch(cmd) {
    ......//关于 GET 的操作就不用看了
57.     case SIOCSIFFLAGS:
58.         if (colon) {
59.             ret = -EADDRNOTAVAIL;
60.             if (!ifa)
61.                 break;
62.             ret = 0;
63.             if (!(ifr.ifr_flags & IFF_UP))
64.                 inet_del_ifa(in_dev, ifap, 1);
65.             break;
66.         }
67.         ret = dev_change_flags(dev, ifr.ifr_flags);
68.         break;
69.
70.     case SIOCSIFADDR:   /* 设置接口地址和地址族 */
71.         ret = -EINVAL;
72.         if (inet_abc_len(sin->sin_addr.s_addr) < 0)
73.             break;
74.
75.         if (!ifa) {
76.             ret = -ENOBUFS;
         申请一个 in_ifaddr{}结构，并且初始化为 0
77.             if ((ifa = inet_alloc_ifa()) == NULL)
78.                 break;
         如果有冒号，就把用户设置的设备名复制到 ifa 中，否则就用设备缺省的名字如“eth0”
79.             if (colon)
80.                 memcpy(ifa->ifa_label, ifr.ifr_name, IFNAMSIZ);
81.             else
82.                 memcpy(ifa->ifa_label, dev->name, IFNAMSIZ);
83.         } else {
84.             ret = 0;
    根据 32 行，我们得到了 ifa，如果用户输入的地址和已经存在的地址一样就跳出 switch
85.             if (ifa->ifa_local == sin->sin_addr.s_addr)
86.                 break;
    如果不一样，我们要删掉老的地址，记住我们要传入的是原本就有的 ifa
87.             inet_del_ifa(in_dev, ifap, 0);
88.             ifa->ifa_broadcast = 0;
89.             ifa->ifa_anycast = 0;
90.         }
91.
92.       ifa->ifa_address = ifa->ifa_local = sin->sin_addr.s_addr;
93.
94.       if (!(dev->flags & IFF_POINTOPOINT)) {
         根据 IP 地址获取网络掩码长度，其值是 0,8,16,24，然后在根据其设置网络掩码
95.          ifa->ifa_prefixlen = inet_abc_len(ifa->ifa_address);
```

```
96.         ifa->ifa_mask = inet_make_mask(ifa->ifa_prefixlen);
97.         if ((dev->flags & IFF_BROADCAST) && ifa->ifa_prefixlen < 31)
98.             ifa->ifa_broadcast = ifa->ifa_address | ~ifa->ifa_mask;
99. } else {
        如果是点到点设备，其掩码是 32 位，即全 1
100.            ifa->ifa_prefixlen = 32;
101.            ifa->ifa_mask = inet_make_mask(32);
102.        }
103.        ret = inet_set_ifa(dev, ifa);
104.        break;
105.
106.    ......
107.    case SIOCSIFNETMASK:   /* Set the netmask for the interface */
108.        /*
109.         *  The mask we set must be legal.
110.         */
111.        ret = -EINVAL;
112.        if (bad_mask(sin->sin_addr.s_addr, 0))
113.            break;
114.        ret = 0;
115.        if (ifa->ifa_mask != sin->sin_addr.s_addr) {
116.            inet_del_ifa(in_dev, ifap, 0);
117.            ifa->ifa_mask = sin->sin_addr.s_addr;
118.            ifa->ifa_prefixlen = inet_mask_len(ifa->ifa_mask);
119.            inet_insert_ifa(ifa);
120.        }
121.        break;
122.    }
123.    done:
124.         ......;几乎什么也不做，就退出了
125.    out:
126.        return ret;
127.    rarok:
128.
129.        ret = copy_to_user(arg, &ifr, sizeof(struct ifreq)) ? -EFAULT : 0;
130.        goto out;
131.    } // 函数结束
```

代码段 3-5　devinet_ioctl 函数

在上面的 switch 语句 SIOCSIFADDR 分支中我们分配了 in_ifaddr{}结构，然后把它传入下面这个函数 inet_set_ifa，它的基本内容是申请 in_dev{}结构，然后把它挂到 ifa 结构中，并把这个 ifa 存入其源代码如下：

```
1.  static int inet_set_ifa(struct net_device *dev, struct in_ifaddr *ifa)
2.  {
3.      struct in_device *in_dev = __in_dev_get(dev);
4.
5.      if (!in_dev) {
6.          in_dev = inetdev_init(dev);
7.          ......
8.      }
9.      if (ifa->ifa_dev != in_dev) {
10.         ifa->ifa_dev = in_dev;
11.     }
        如果是 loopback 地址，那么该地址的 scope 是本机范围，否则还是初始值 0，即 UNIVERSE。
12.     if (LOOPBACK(ifa->ifa_local))
13.         ifa->ifa_scope = RT_SCOPE_HOST;
14.     return inet_insert_ifa(ifa);
15. }
```

代码段 3-6　inet_set_ifa 函数

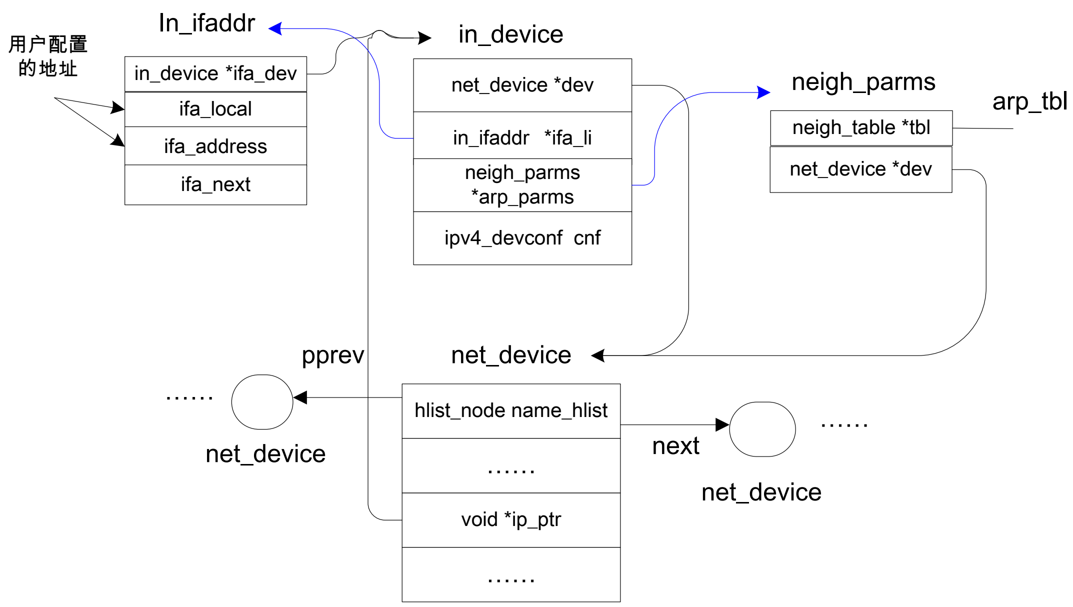
如果在该设备上没有找到相应的 IP 地址配置，就创建一个 in_device 结构。

```
1.  struct in_device *inetdev_init(struct net_device *dev)
2.  {
```

```
3.       struct in_device *in_dev;
4.
5.       in_dev = kmalloc(sizeof(*in_dev), GFP_KERNEL);
6.       ……
7.       memset(in_dev, 0, sizeof(*in_dev));
8.
9.       memcpy(&in_dev->cnf, &ipv4_devconf_dflt, sizeof(in_dev->cnf));
10.      in_dev->cnf.sysctl = NULL;
11.      in_dev->dev = dev;
         创建一个邻居参数，而且挂到 arp_tbl 上，我们会放到 ARP 的一章中介绍
12.      if ((in_dev->arp_parms = neigh_parms_alloc(dev, &arp_tbl)) == NULL)
13.          goto out_kfree;
14.      inet_dev_count++;
15.      /* Reference in_dev->dev */
16.
         将 ip_ptr 指向 in_dev
17.      dev->ip_ptr = in_dev;
18.
19.    out:
20.      return in_dev;
21.    out_kfree:
22.      kfree(in_dev);
23.      in_dev = NULL;
24.      goto out;
25.  }
```

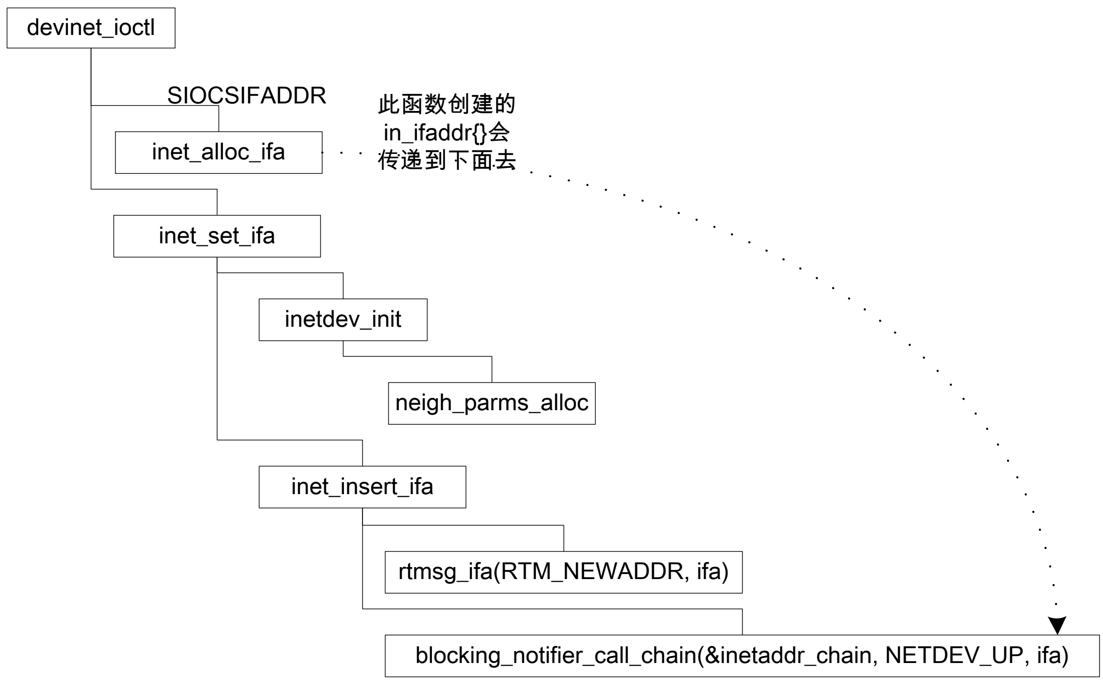
代码段 3-7 inetdev_init 函数

经过这么一番配置，in_ifaddr{}, net_device{}和 in_device{}的关系如下图，而且要记住，in_ifaddr{}这个结构里的成员不会被其它模块改变了：



图表 3-8 inet_set_ifa 之后数据结构之间的关系

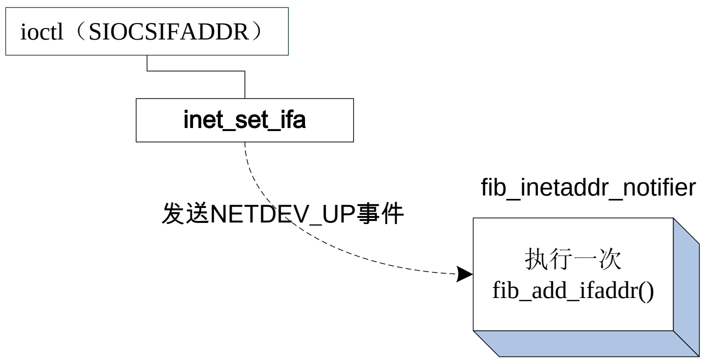
经过上面的分析，代码的大致流程如下：



图表 3-9 devinet_ioctl 函数调用树

首先调用 rtmsg_ifa 发送消息给应用层，以至于 netlink 的侦听函数会知道这个新地址，然后调用通知链，这会触发对地址分配 IP 事件感兴趣的等候者行动。

对 ioctl 分析也快到头了，但好像还没看到跟路由有什么关系，难道线索断了？但是我们忘了，**blocking_notifier_call_chain** 扫描的是 inetaddr_chain，顾名思义，这是一条链呀，对了，还记得在 FIB 初始化的时候，它曾经也挂到了这个链上吗？



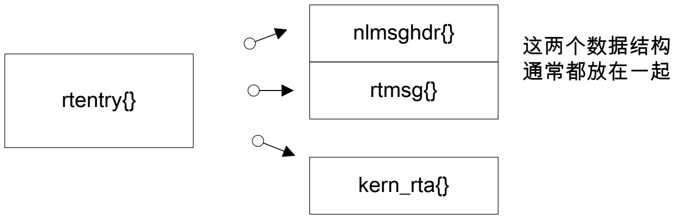
图表 3-10 inet_set_ifa 发送 NETDEV_UP 事件

3.1.3.2. netlink 和 rtnetlink 接口

虽然我们还没有说到路由系统，但是为了解释刚才看见的 netlink_broadcast 函数，没办法。通常，在 Linux 里当一个应用程序想要增加或删除一个路由时，它会用 nlmsghdr+rtmsg 结构并传给 rtnetlink socket。

rtentry 结构是 UNIX 系统提供给用户接口获取路由的传统途径，实际上 Linux 内部并不使用这个结构，它和 BSD 操作系统用作路由 ioctl 的 rtentry 相似，只是用来方便 Linux 和其他 UNIX 变种的移植。一个指向 rtentry 的指针作为参数传到 ioctl 系统调用，然后通过 fib_convert_rtentry

（int cmd, struct nlmsghdr *nl, struct rtmsg *rtm, struct kern_rta *rta, struct rtentry *r)函数把这个结构的成员分拆到 nlmsghdr{}、rtmmsg{}、kern_rta{}结构中。



图表 3-11 rtentry 被拆分成 3 个部分

访问 FIB 的主要方法是通过 netlink socket，它为路由提供了扩展——rtnetlink。netlink 是一个内部的通信协议。它主要用来在应用层和 Linux 内核里大量协议之间传递信息。Netlink 实现了自己的地址族——AF_NETLINK，它支持大多数的 socket API 函数。netlink 最常用的场合是应用程序预内核内部的路由表交换路由信息。其使用方法是先打开一个 socket：fd = socket(PF_NETLINK, socket_type, netlink_family);socket_type 为 SOCK_RAW 或者 SOCK_DGRAM 都可以，因为 netlink 本身是基于数据报的。比较常用 netlink_family 有下面几种：

- NETLINK_ROUTE

  用来修改和读取路由表的，这是我们后面要讨论的 rtnetlink 问题.

- NETLINK_ARPD

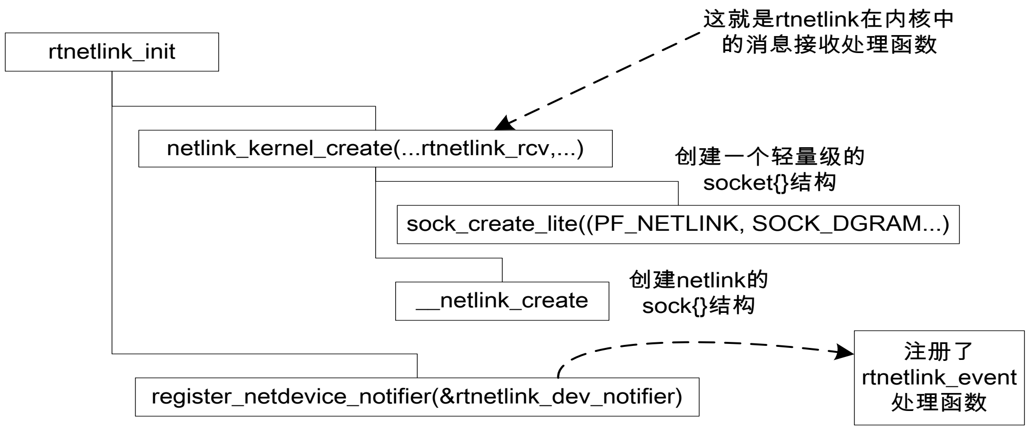
  在用户空间中管理 ARP 表.

- NETLINK_USERSOCK

  给应用程序（不一定都是协议）发送的消息

打开了 socket，那么可以用 sendmsg 和 recvmsg 给内核或同台主机应用程序（而不是发给远端主机）发消息。这些调用传递一个指向 nlmsghdr 结构的指针（见上图）。

rtnetlink 是基本 netlink 协议的消息扩展。也就是当创建 PF_NETLINK 的 socket 时，把 netlink_family 设置为 NETLINK_ROUTE。内核创建了一个轻量级的 socket，专门用于接收来自用户 netlink 接口发送过来的消息，也接收内核部分的消息。



图表 3-12 rtnetlink_init 函数调用树

内核模块使用 netlink_broadcast 给 rtnetlink 发送消息，由 rtnetlink_rcv 函数接收并处理。

下面继续解说 inet_insert_ifa 函数，它里面利用了一个 C 语言技巧，首先它创建一个 skb，然后进入一个判断语句，首先判断 inet_fill_ifaddr 函数执行的结果，在大部分情况下，必定会进入到最后一个 else 语句，执行 netlink_broadcast，这种技巧在内核代码中比较常见，而且通常能造成读者迷惑，认为最后一句 else 的情况不会到达。这种技巧的目的在于遍历各种情况，而且不影响代码的“从上到下”顺序。

```
1.  static void rtmsg_ifa(int event, struct in_ifaddr* ifa)
2.  {
3.      int size = NLMSG_SPACE(sizeof(struct ifaddrmsg) + 128);
4.      struct sk_buff *skb = alloc_skb(size, GFP_KERNEL);
5.
6.      if (inet_fill_ifaddr(skb, ifa, current->pid, 0, event, 0) < 0) {
        错误处理
7.      } else {
    下面这个函数会通知阻塞在某个 netlink 下的回调函数，然后让它们进行处理。不过要注意，rtnl 是
    一个 sock{}结构的全局变量
8.          netlink_broadcast(rtnl, skb, 0, RTNLGRP_IPV4_IFADDR, GFP_KERNEL);
9.      }
10. }
```

代码段 3-8 rtmsg_ifa 函数

当 rtnetlink_rcv 接收到这样一个消息，它会根据参数调用对应的函数表。在这里它会调用 inet_rtm_newaddr 函数。inet_rtm_newroute 增加一个新的路由到 FIB。inet_rtm_delroute 从 FIB 中删除一条路由。这两个函数从紧跟 nlmsghdr 后面的内存中抽取 rtmsg 结构，rtmsg 的结构如下：

```
struct rtmsg
{
   uchar rtm_family;
   下面两个是，用来给 AF_INET 类型的地址创建 32 位或更小的网络掩码 bit 的位数
   uchar rtm_dst_len;
   uchar rtm_src_len;
   uchar rtm_tos; //对应 IP 头部的 ToS 字段
   uchar rtm_table;//包含路由表 ID，如果没有配置多个表，那么就是 RT_TABLE_MAIN
   或 RT_TABLE_LOCAL.
   uchar rtm_protocol;//指的是下表的指
```

表格 3-2

Protocol	Value	Purpose
RTPROT_UNSPEC	0	The value is unspecified.
RTPROT_REDIRECT	1	这条路由是由 ICMP 重定向消息产生的,IPv4 当前没有使用
RTPROT_KERNEL	2	这条路由是内核产生的
RTPROT_BOOT	3	这条路由是在 boot 的过程中产生的
RTPROT_STATIC	4	这跳路由是管理员静态设置的
RTPROT_GATED	8	这是由网关路由守护进程使用
RTPROT_RA	9	Used for RDISC/ND router advertisements.
RTPROT_MRT	10	This value is used by Merit MRT.
RTPROT_ZEBRA	11	由 Zerba 路由守护进程使用（这个 Zerba 比较有名）
RTPROT_BIRD	12	Used by BIRD.
RTPROT_DNROUTED	13	Used by DECnet routing daemon.

```
   uchar rtm_scope;
   uchar rtm_type;//路由类型
   uint Rtm_flags;//可以是下表 3 个值
```

```
表格 3-3
```

宏	值	意义
RTM_F_NOTIFY	x100	通知用户路由改变
RTM_F_CLONED	0x200	指示路由被 clone 了
RTM_F_EQUALIZE	0x400	还没有被实现。。。。

```
}
```

代码段 3-9 rtmsg 机构

Rtmsg 结构的协议字段的值如果大于 RTPROT_STATIC，那么内核不会改变它，只是在用户和内核空间传递。它们故意留着给假想的几个路由守护进程使用。代码里的注释推荐这些值应该标准化以避免冲突。

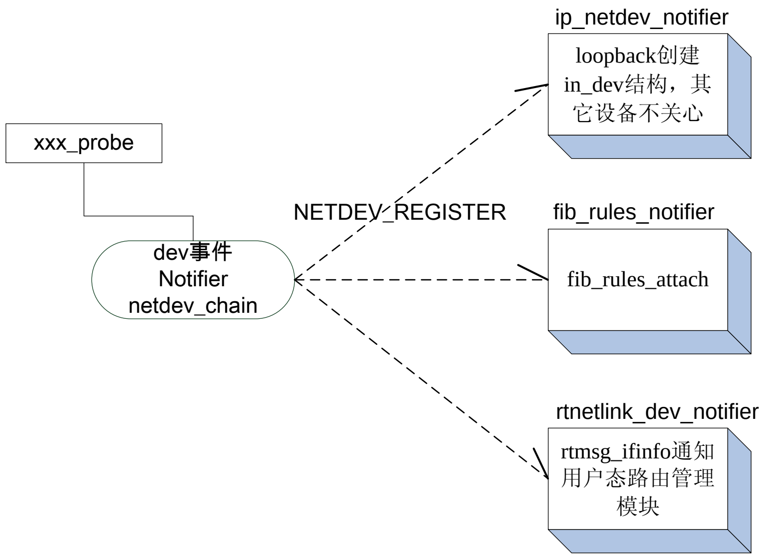
3.1.4 Loopback 接口的配置过程

前面一节介绍了给本机系统配置 IP 地址的过程，这一节给大家介绍一下 loopback 接口的“配置”过程，之所以用引号，是因为此配置不完全是用户自己控制的，为什么不先介绍 loopback 的配置，原因也在此。上一节我们已经对配置的过程一步一步做了分解，那么我们可以一下子来了解 loopback 接口的初始化及配置过程，这也是对普通设备的初始化和配置过程的一个回顾。

要使 Loopback 接口起作用得分为 2 个步骤：

第一步是在系统初始化的时候就创建一个 in_dev{}结构给 loopback 接口，参照上一节，对于用户配置普通设备的 IP 地址，这个动作应该是在 ioctl→...... →inet_set_ifa 函数中完成。这是两种不同设备的配置上的区别：loopback 设备是自动创建，而普通设备是“用户”手动创建。

而其这种自动还转了一道手，它实际是在 ip_netdev_notifier 收到 NETDEV_REGISTER 事件后才去调用 inetdev_init 创建了 in_dev{}结构，此函数请参考前一节。而普通设备则是在 ioctl 的上下文中调用 inet_set_ifa 函数创建的。



图表 3-13 probe 发起 NET_DEV_REGISTER 事件

第二个步骤就是当系统初始化结束之后根据系统配置打开 loopback 接口，使之 UP。

如同普通设备的操作，系统必须调用 dev_open 打开 looback 接口，从而发送 NETDEV_UP 事件给 ip_netdev_notifier。这个 notifier 对于普通设备的事件还是不是太感兴趣，啥也不做；但是非

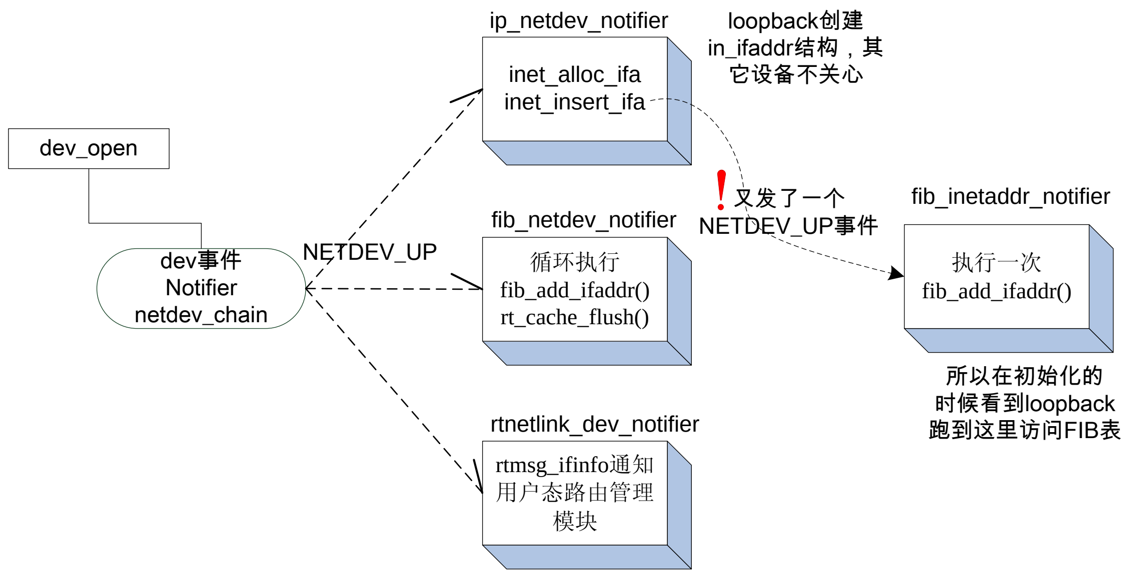
常关心 loopback 接口的 UP 事件，它创建了 in_ifaddr{}结构。这有是 loopback 接口配置自动化的证明了。

ip_netdev_notifier 的事件处理函数是 inetdev_event，其相关代码如下：

```
1.  inetdev_event
2.  {
3.  ......
4.  case NETDEV_UP:
5.      if (dev == &loopback_dev) {
6.          struct in_ifaddr *ifa;
7.          if ((ifa = inet_alloc_ifa()) != NULL) {
              这里面每个赋值都很重要，直接决定 loopback 的性质
              INADDR_LOOPBACK 宏就是 0x7f000001  /* 127.0.0.1   */
8.              ifa->ifa_local = ifa->ifa_address = htonl(INADDR_LOOPBACK);
9.              ifa->ifa_prefixlen = 8;
10.             ifa->ifa_mask = inet_make_mask(8);
11.
12.             ifa->ifa_dev = in_dev;
               该地址是属于本机范围的，不属于外部地址，我们会在后面说到这个值
13.             ifa->ifa_scope = RT_SCOPE_HOST;
14.             memcpy(ifa->ifa_label, dev->name, IFNAMSIZ);
15.             inet_insert_ifa(ifa);
16.         }
17.     }
18. ......
19. }
```

代码段 3-10 loopback 设备对 NETDEV_UP 事件的处理

值得关注的是这段代码又调用了 inet_insert_ifa 函数，它发送 NETDEV_UP 事件给 fib_inetaddr_notifier。



图表 3-14 dev_open 发起 NETDEV_UP 事件

是否 loopback 设备的配置可以改变？可以的！让我们跑到 Linux 的/etc/sysconfig/ network 目录下（目前是 Redhat，其它版本的 Linux 不一定是此目录，但都大同小异），看到有 ifcfg-lo 这么一个文件，内容如下：

```
DEVICE=lo
IPADDR=127.0.0.1          # 可以改变这个 IP 地址
NETMASK=255.0.0.0
NETWORK=127.0.0.0
......
BROADCAST=127.255.255.255
NAME=loopback
```

代码段 3-11 ifcfg-lo 文件里的内容

你可以改动这个文件里的某些项，然后执行**#ifup lo** 就把里面的内容作为 ioctl 的参数传给了内核，把缺省的配置改成你想要的。

上图中看到 fib_netdev_notifier 收到 NETDEV_UP 事件（确切的来说此时只有 loopback 接口），于搜索 in_dev→fa_list 然后循环调用 fib_add_ifaddr，实际在缺省情况下在系统启动之后，只有 loopback 接口才有 in_ifaddr{}结构，所以这个动作对于普通设备没有意义。我们将在下面两节介绍 fib_add_ifaddr 函数，这个函数非常重要，它是打开路由系统的钥匙，请读者耐心等候，稍安勿燥。

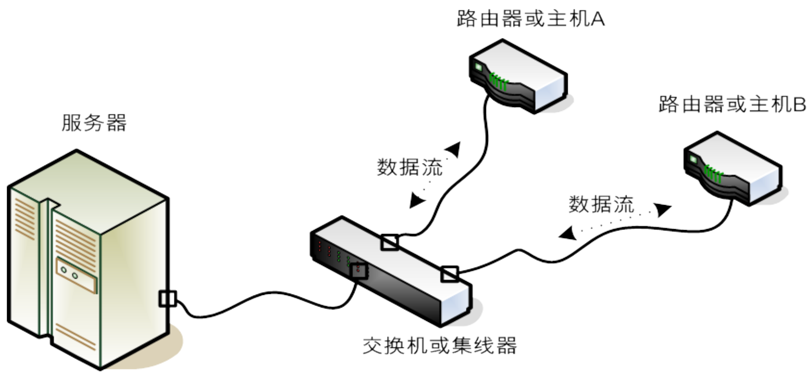
3.1.5 IP 别名的实现

上一节说的是用 ifconfig 命令来进行接口的地址配置，这个命令已经廉颇老矣，更先进的配置工具是 ip 命令。在本节中我们将介绍一个叫做 IP 别名的概念。为了能处理 IP 别名，所以要引入 ip 命令。

ip 是 iproute2 软件包里面的一个强大的网络配置工具，Iproute2 是一个在 Linux 下的高级网络管理工具软件。实际上，它是通过 rtnetlink sockets 方式动态配置内核的一些小工具组成的，从 Linux2.2 内核开始，就实现了通过 rtnetlink sockets 用来配置网络协议栈，它是一个现代的强大的接口。最吸引人的特色就是它用完整而有机制的简单命令替代了之前以下命令的功能，如 ifconfig,arp,route,iptunnel，而且还添加了其它不少的功能。如今，Iproute2 已经在很多主要的发行版里被默认安装。它在配置隧道的时候非常有用。当然本书不打算讨论它们。我们要讨论的 IP 别名，通过 iproute2 这个工具可以对它进行设置。

前面已经零零散散的说到了 netlink 技术，现在我们又要介绍一个和它相关的工具——ip 命令。在剩下的章节中我们会常看到 ip 命令的使用。

IP 别名有时候也称为网络接口别名（network interface aliasing） 或逻辑接口（logical interface）。IP 别名的概念是：可以在一个网络接口上配置多个 IP 地址。这样就能够在使用单一接口的同一个主机上进行负载平衡以支持多种服务。比如你的主机上只有少量网卡接口，而又要求进行多路接入的扩展，你可以如下配置：



图表 3-15 IP 别名的用途

假设 A 和 B 与服务器之间的数据不相关，在这种配置下，你的服务器为了给两个不同的请求服务，要么使机器上的端口（指 TCP/UDP 的端口，比如 FTP 使用的 21，HTTP 使用 80）不一样，要么使机器上的 IP 地址不一样——如果大家都要使用 FTP，而且请求端口还是 21，那只好使用多个逻辑 IP 了。当然读者还有更多的解决方法，但本书提出这样一个概念只是告诉你 Linux“可以”用这种解决办法，而非“必须”。

例如，如果在物理单元号为 eth0 的以太网卡上已经配置了一个现有的 IP 地址，那么可以通过添加一个逻辑单元号 :1 来创建 IP 别名，可以通过递增逻辑单元号来添加更多的 IP 地址。

配置的基本过程如下：

假设我们已经有一个 IP 地址了，如上一节所配 192.168.18.2。

ip addr add 192.168.0.11/24 label eth0:1 dev eth0

检查一下是否设置成功

ip addr show eth0

2: eth0: mtu 1500 qdisc pfifo_fast qlen 100

link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff

inet 192.168.18.2/24 brd 192.168.18.255 scope global eth0

inet 192.168.0.11/24 scope global secondary eth0:1

ip addr 命令和 ifconfig 在内核的实现路径不一致，后者是通过 ioctl，前者则是通过 netlink 消息接口，内核中采用 inet_rtm_newaddr 来响应这个消息。此函数把应用层传过来的参数进行一次拷贝之后，创建 in_ifaddr{}，就直接调用 inet_insert_ifa。

```
1.  static int inet_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)
2.  {
3.  struct rtattr **rta = arg;
4.  struct net_device *dev;
5.  struct in_device *in_dev;
6.  struct ifaddrmsg *ifm = NLMSG_DATA(nlh);
7.  struct in_ifaddr *ifa;
8.  int rc = -EINVAL;
9.
10. ASSERT_RTNL();
11.
12. if (ifm->ifa_prefixlen > 32 || !rta[IFA_LOCAL - 1])
13.     goto out;
14.
```

```
15. dev = __dev_get_by_index(ifm->ifa_index);
16.
17. rc = -ENOBUFS;
18. if ((in_dev = __in_dev_get_rtnl(dev)) == NULL) {
19.     in_dev = inetdev_init(dev);
20.      .....
21. }
22.
23. ifa = inet_alloc_ifa();
24.
25. if (!rta[IFA_ADDRESS - 1])
26.     rta[IFA_ADDRESS - 1] = rta[IFA_LOCAL - 1];
27. memcpy(&ifa->ifa_local, RTA_DATA(rta[IFA_LOCAL - 1]), 4);
28. memcpy(&ifa->ifa_address, RTA_DATA(rta[IFA_ADDRESS - 1]), 4);
29. ifa->ifa_prefixlen = ifm->ifa_prefixlen;
30. ifa->ifa_mask = inet_make_mask(ifm->ifa_prefixlen);
31. if (rta[IFA_BROADCAST - 1])
32.     memcpy(&ifa->ifa_broadcast,
33.            RTA_DATA(rta[IFA_BROADCAST - 1]), 4);
34. if (rta[IFA_ANYCAST - 1])
35.     memcpy(&ifa->ifa_anycast, RTA_DATA(rta[IFA_ANYCAST - 1]), 4);
36. ifa->ifa_flags = ifm->ifa_flags;
37. ifa->ifa_scope = ifm->ifa_scope;
38.
39. ifa->ifa_dev   = in_dev;
40. if (rta[IFA_LABEL - 1])
41.     rtattr_strlcpy(ifa->ifa_label, rta[IFA_LABEL - 1], IFNAMSIZ);
42. else
43.     memcpy(ifa->ifa_label, dev->name, IFNAMSIZ);
44.
45. rc = inet_insert_ifa(ifa);
46. out:
47. return rc;
48. }
```

代码段 3-12 inet_rtm_newaddr 函数

这段代码是不是和 inet_set_ifa 函数很象啊？

网络设备管理层已经为这个机制做好了准备。让我们来看看曾经走过的路。inet_insert_ifa 之前就看到了，只是没有给出它的详细代码，为了要弄清楚 IP 别名在内核中的实现，那就得看详细代码：

```
49. static int inet_insert_ifa(struct in_ifaddr *ifa)
50. {
        在这里 ifa 就是根据应用层分配的地址而创建的内核地址结构
51.     struct in_device *in_dev = ifa->ifa_dev;
52.     struct in_ifaddr *ifa1, **ifap, **last_primary;
53.     ......
        假设该 ifa 是该地址的第一个地址，而不是第二个
54.     ifa->ifa_flags &= ~IFA_F_SECONDARY;
55.     last_primary = &in_dev->ifa_list;
        开始遍历设备的地址列表
56.     for (ifap = &in_dev->ifa_list; (ifa1 = *ifap) != NULL;
57.         ifap = &ifa1->ifa_next)
58.     {
            如果链上的地址不是第二个地址并且 scope 小于等于列表上的 scope，last 指向下一个地址节点
59.         if (!(ifa1->ifa_flags & IFA_F_SECONDARY) &&
60.             ifa->ifa_scope <= ifa1->ifa_scope)
61.              last_primary = &ifa1->ifa_next;
62.         if (ifa1->ifa_mask == ifa->ifa_mask &&
63.             inet_ifa_match(ifa1->ifa_address, ifa))
64.         {
                如果两个地址掩码相同而且是同一子网，那么：
                如果两个地址一样就退出
65.              if (ifa1->ifa_local == ifa->ifa_local) {
66.                  inet_free_ifa(ifa);
```

```
67.                 return -EEXIST;
68.             }
               如果两个地址的 scope 不一样也退出
69.             if (ifa1->ifa_scope != ifa->ifa_scope) {
70.                 inet_free_ifa(ifa);
71.                 return -EINVAL;
72.             }
              否则这个地址就是第二地址，也就是说，如果第一个节点不是 SCONDARY，那么后面增加的 IP 地
              址必定是 SECONDARY
73.             ifa->ifa_flags |= IFA_F_SECONDARY;
74.         }
75.         然后继续查找符合条件的地址节点，
76.     }
77.
78.     if (!(ifa->ifa_flags & IFA_F_SECONDARY)) {
79.         net_srandom(ifa->ifa_local);
80.         ifap = last_primary;
81.     }
       如果设备上的地址链表是空的，就把它挂在后面，如果不是空的，则根据上面的判断把地址放到合适的位置。
82.     ifa->ifa_next = *ifap;
83.     *ifap = ifa;
84.     rtmsg_ifa(RTM_NEWADDR, ifa);
85.     blocking_notifier_call_chain(&inetaddr_chain, NETDEV_UP, ifa);
86.
87.     return 0;
88.}
```

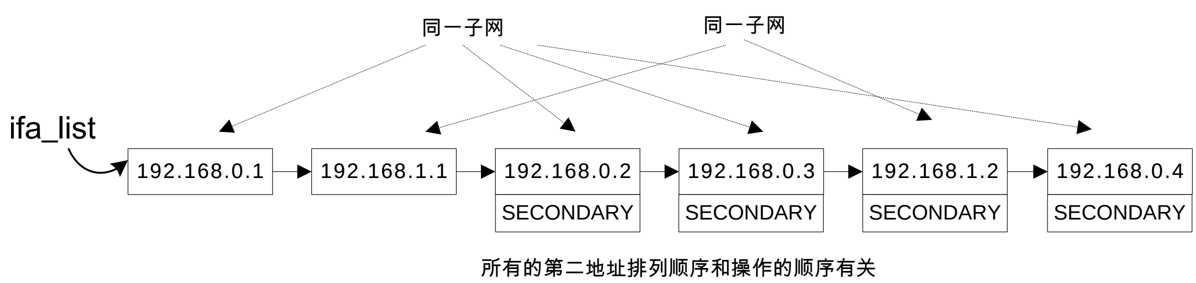
代码段 3-13 inet_insert_ifa 函数

这里要注意的是 in_ifaddr→local 和 in_ifaddr→address 在绝大多数情况下是一样的，目前我没有找到不是一样的例子。有哪位读者可以告诉我这两者有什么区别吗？

上面那段代码完成的工作就是把该接口上的 IP 地址组成一条链表，如果我们在一台机器上为某块网卡设置 5 个 IP 地址，其配置顺序如下：

```
# ip addr add 192.168.0.1/24 label eth0:1 dev eth0
# ip addr add 192.168.0.2/24 label eth0:2 dev eth0
# ip addr add 192.168.0.3/24 label eth0:3 dev eth0
# ip addr add 192.168.1.1/24 label eth0:4 dev eth0
# ip addr add 192.168.1.2/24 label eth0:5 dev eth0
# ip addr add 192.168.0.4/24 label eth0:6 dev eth0
```

那么内核中 in_dev→ifa_list 的组成形式如下：

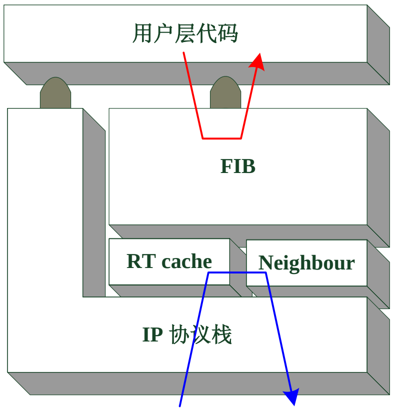


图表 3-16 ifa_list 的组织形式

当配置最后一个地址时，它还是挂在链表最后。

3.2 回顾 FIB 系统初始化

协议栈的初始化前一章已经大体介绍了，当时把路由模块的初始化跨过时因为这个模块最好能一次讲清楚，如果分开来说会给人零散的感觉。为什么要从配置 IP 地址的章节开始呢？是因为在配置 IP 地址时内核就开始存取 FIB 系统了。初始化只是搭起了一个架子，还只是一具空壳，完全看不出 FIB 系统是如何组织和工作的。在 Linux 路由系统中主要保存了三种与路由相关的数据，第一种是在物理上和本机相连接的主机地址信息表——相邻表：neigh_table{ }，第二种是保存了在网络访问中判断一个网络地址应该走什么路由的数据表——路由规则表：fib_table{ }，第三种表是最新使用过的查询路由地址的缓存地址数据表——路由缓存：rtcache， 由 rtable{ }节点组成。它们三者之间的关系如下图：



图表 3-17 FIB 和 RT cache 的关系

上图中 FIB 主要是和用户层打交道，而 RT cache 主要是和协议栈打交道，除非处于维护的目的，协议栈是不会去直接访问 FIB 系统。在一个基本稳定系统中，红线箭头表示用户操作流，蓝线箭头表示底层报文的路由查询过程。只有特殊情况，报文路由过程才会进入到 FIB 系统查询，这个原理和 CPU 访问 Cache 不命中然后才访问内存是一个道理。

现在回过来看 ip_init，它一上来就调用 ip_rt_init，为 FIB 搭好一个空架子：

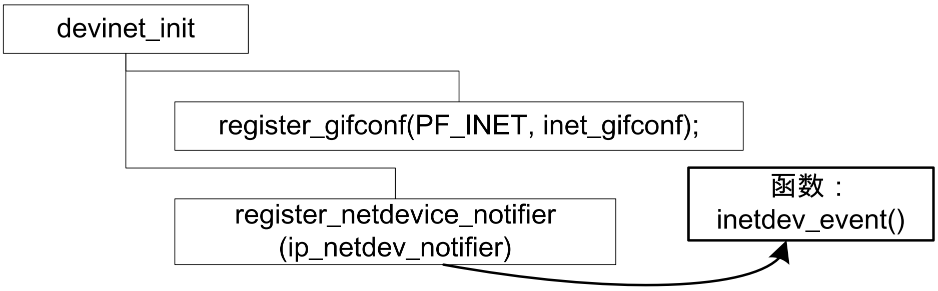
```
20.
21. int __init ip_rt_init(void)
22. {
23.     int rc = 0;
24.
25.     rt_hash_rnd = (int) ((num_physpages ^ (num_physpages>>8)) ^
26.                 (jiffies ^ (jiffies >> 7)));
27.
28.     #ifdef CONFIG_NET_CLS_ROUTE
29.     {
30.     int order;
31.     for (order = 0;
32.          (PAGE_SIZE << order) < 256 * sizeof(struct ip_rt_acct) * NR_CPUS; order++)
33.         /* NOTHING */;
34.     ip_rt_acct = (struct ip_rt_acct *)__get_free_pages(GFP_KERNEL, order);
35.     memset(ip_rt_acct, 0, PAGE_SIZE << order);
36.     }
37.     #endif
38.
39.     ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
40.                           sizeof(struct rtable),
41.                           0, SLAB_HWCACHE_ALIGN,
```

```
42.                             NULL, NULL);
43.
        缺省情况下 rt_hash_table 的表项是 4000 条，那么占用内存 16000 字节。分配内存的这个函数我们
        会在 TCP 初始化的代码中再次看到，我们这里就不介绍了
44.     rt_hash_table = (struct rt_hash_bucket *)
45.         alloc_large_system_hash("IP route cache",
46.                     sizeof(struct rt_hash_bucket),
47.                     rhash_entries,
48.                     (num_physpages >= 128 * 1024) ?
49.                     15 : 17,
50.                     0,
51.                     &rt_hash_log,
52.                     &rt_hash_mask,
53.                     0);
54.     memset(rt_hash_table, 0, (rt_hash_mask + 1) * sizeof(struct rt_hash_bucket));
55.     rt_hash_lock_init();
56.
57.     ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
58.     ip_rt_max_size = (rt_hash_mask + 1) * 16;
59.
60.     devinet_init();
61.     ip_fib_init();
62.
63.     init_timer(&rt_flush_timer);
64.     rt_flush_timer.function = rt_run_flush;
65.     init_timer(&rt_periodic_timer);
66.     rt_periodic_timer.function = rt_check_expire;
67.     init_timer(&rt_secret_timer);
68.     rt_secret_timer.function = rt_secret_rebuild;
69.
70.     /* All the timers, started at system startup tend
71.        to synchronize. Perturb it a bit.
72.      */
73.     rt_periodic_timer.expires = jiffies + net_random() % ip_rt_gc_interval +
74.                     ip_rt_gc_interval;
75.     add_timer(&rt_periodic_timer);
76.
77.     rt_secret_timer.expires = jiffies + net_random() % ip_rt_secret_interval +
78.         ip_rt_secret_interval;
79.     add_timer(&rt_secret_timer);
80.
81.     #ifdef CONFIG_PROC_FS
82.     {
83.     struct proc_dir_entry *rtstat_pde = NULL; /* keep gcc happy */
84.     if (!proc_net_fops_create("rt_cache", S_IRUGO, &rt_cache_seq_fops) ||
85.         !(rtstat_pde = create_proc_entry("rt_cache", S_IRUGO,
86.                         proc_net_stat))) {
87.         return -ENOMEM;
88.     }
89.     rtstat_pde->proc_fops = &rt_cpu_seq_fops;
90.     }
91.     #ifdef CONFIG_NET_CLS_ROUTE
92.     create_proc_read_entry("rt_acct", 0, proc_net, ip_rt_acct_read, NULL);
93.     #endif
94.     #endif
95.     return rc;
96. }
```

代码段 3-14 ip_rt_init 函数

在这个函数中涉及到 2 个刚才提及的表的初始化：FIB 表和路由表，但没有见到邻居表的初始化，那邻居表在哪初始化的呢？

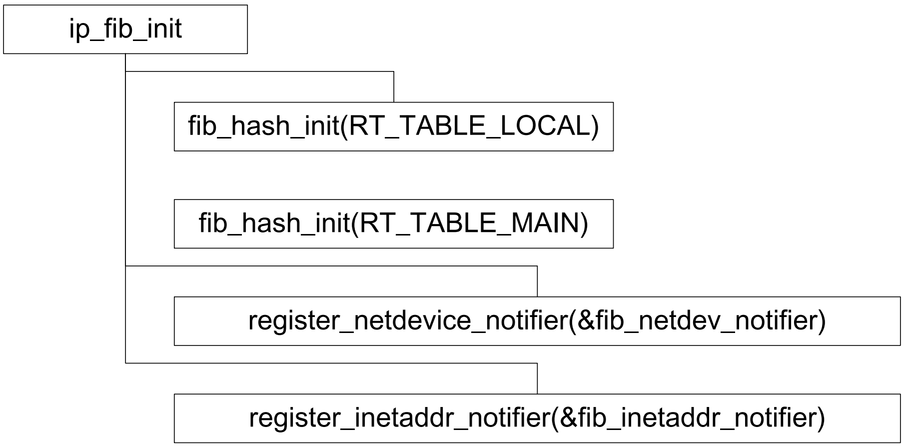
devinet_init( )的流程较简单，图示如下：



图表 3-18　devinet_init 函数调用树

此函数主要是注册两个事件通知回调函数，前面一个回调函数对于本书没有太多意义，所以不会介绍；而第二个函数注册的是一个叫做 ip_netdev_notifier 的通知块，其处理函数是 inetdev_event，即产生某些事件的时候，会以通知的形式回调此函数。让我回想一下在设备接口初始化例程中第 3 步和第 4 步我们有两次发送事件，一次是发送 NETDEV_REGISTER，一次是 NETDEV_UP 事件，这两个事件都会引起 inetdev_event 的调用，对此回调函数的讲解我放到下一章了，在此读者只需知道我们挂接了该回调即可。

那下面要说的跟 fib 相关的函数虽然重要，但内部操作并不复杂。



图表 3-19ip_fib_init 函数调用树

ip_fib_init 注册了 2 个通知块，读者们可要看清楚了（我当初就因为眼花而没有注意到这个差别），一个是挂到 netdev_chain 上，前面介绍的 devinet_init 函数也挂接了一个通知块在上面；另一个是挂到了 inetaddr_chain 上，也就是说，fib 系统对设备相关的事件和 ip 地址相关事件都感兴趣，我们会在下面的章节中对其进行解释。回过去来看 fib_hash_init，它的作用看似也简单，就是申请一块大内存，还指定这块内存的函树指针，代码如下：

```
1.  struct fib_table * __init fib_hash_init(int id)
2.  {
3.      struct fib_table *tb;
4.  
5.      if (fn_hash_kmem == NULL)
6.          fn_hash_kmem = kmem_cache_create("ip_fib_hash",
7.                           sizeof(struct fib_node),
8.                           0, SLAB_HWCACHE_ALIGN,
9.                           NULL, NULL);
     创建表项的时候要注意它的大小是 fib_table 本身的 size 加上 fn_hash 的 size
10.     tb = kmalloc(sizeof(struct fib_table) + sizeof(struct fn_hash), GFP_KERNEL);
11.     ......//错误处理
12. 
13.     tb->tb_id = id;
14.     tb->tb_lookup = fn_hash_lookup;
15.     tb->tb_insert = fn_hash_insert;
16.     tb->tb_delete = fn_hash_delete;
17.     tb->tb_flush = fn_hash_flush;
18.     tb->tb_select_default = fn_hash_select_default;
19.     tb->tb_dump = fn_hash_dump;
    在定义 tb_data 的时候，我们不知道其大小，但是在上面创建 tb 的时候已经为其留下了足够的空间。fn_hash
     内部实际上是一个数组，读者可以在“配置系统 3.1.4”章可以看到它真实面目
20.     memset(tb->tb_data, 0, sizeof(struct fn_hash));
21.     return tb;
22. }
```

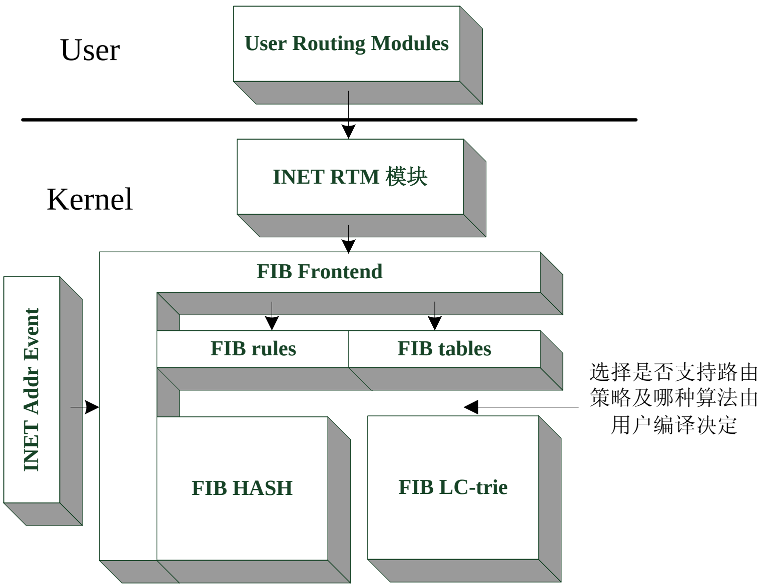
代码段 3-15 fib_hash_init 函数

读者们，这块内存可不简单，它是我们之前提到的 FIB 表，就是所谓的路由数据库，系统的路由信息全部放在里面。ip_fib_init 初始化了 2 个 FIB 表，一个是 local 的，一个是 main 的，到底什么意思，我们在下一节将深入探讨。

3.3 深入 FIB 系统

首先从路由算法说起，因为老学究们一提起路由必谈算法，这里先满足这一部分人。目前的内核路由存在两种查找算法，一种为 HASH 算法，另一种为 LC-trie 算法，在配置内核网络选项的时候选择"IP: advanced router"，然后进入"Choose IP: FIB lookup algorithm (choose FIB_HASH if unsure)"，会看到这两种算法。如果不选择“advanced router”，那么还是使用 hash 算法。Hash 算法是目前 Linux 内核使用的缺省算法，对于大多数应用足够了，而 LC-trie 使用最长前缀匹配算法，在超大路由表的情况下性能比 Hash 算法好，但它大大地增加了算法本身的复杂性和内存的消耗。只可惜本书不能提供更多的关于算法的知识，让老学究们失望了。不过我一定要解释 FIB 表是如何被存取的，这才是现实情况下开发人员最关心的事。

我们注意到 ip_fib_init 函数是放在一个叫做 fib_frontend.c 的文件中，frontend，即前端的意思。也就是说，FIB 系统分为前端和后端，前端部分包含处理一些和外部系统打交道的函数接口（比如 rtnetlink socket），比如初始化、增删路由、接口地址改变等。而后端是 FIB 内部逻辑操作，选择是否支持路由策略及哪种算法由用户在编译内核的时候决定，但改变内核之后，上层/外部应用并不会感知这种变化。



图表 3-20 Linux 内核路由模块结构

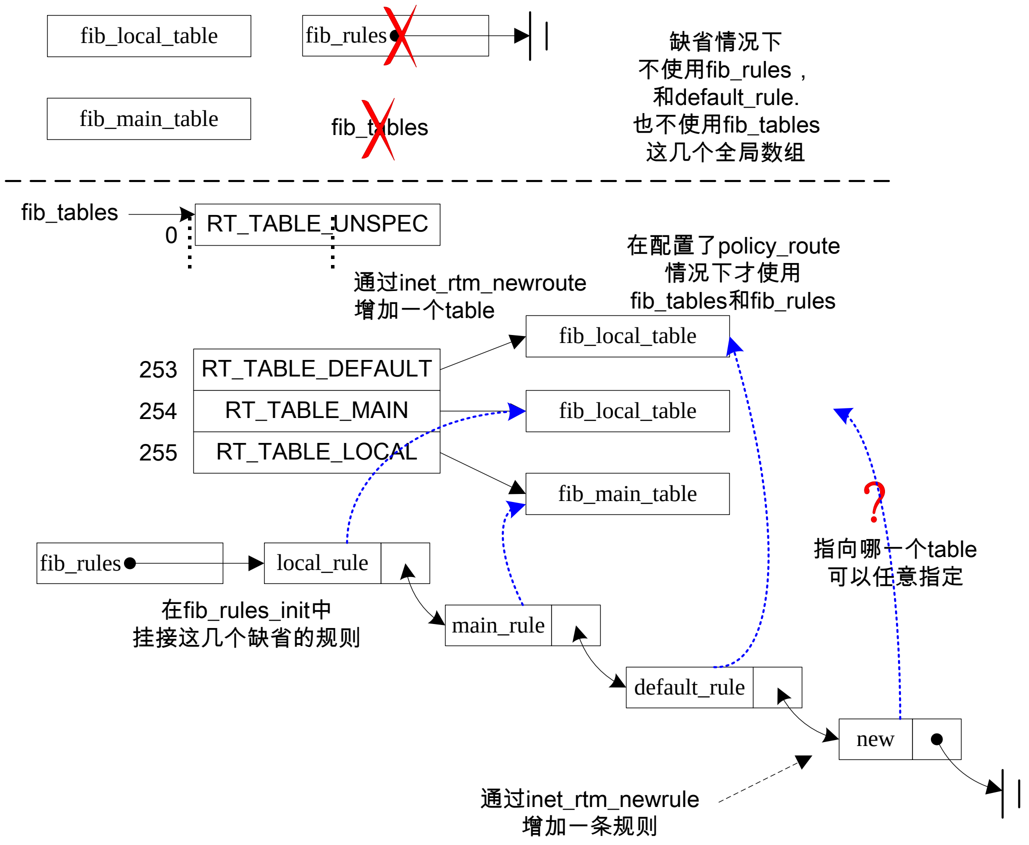
当选择高级路由后，你还可以选择是否支持策略路由（Policy routing），即 Linux 使用了多个路由表来应付有多种路由算法存在的情况，Linux 使用多个路由表，使不同策略的路由存放在不同的表中，有效地被免了查找庞大的路由。即使不使用策略路由，出于性能的考虑，Linux 也使用了两个路由表，一个用于上传给本地上层协议，另一个则用于转发。从效果上看，假设我们有多个路由算法（OSPF，RIP，BGP，静态配置等）同时存在于一台机器上，每个路由算法可以选择不同的 FIB 表作为自己的内核数据库，那么当报文到达 IP 层时选择哪一个数据库作为如何处理（转发？送到本机？还是丢弃）的参考时，就依靠一种规则了。可以是一种算法存取一个 FIB 表，也可以多个算法存取一个 FIB 表。这就是 Policy Route。比如说，对于本地接收的报文应该采用一种规则，那么可以为其单独创建一个 FIB 表；对于组播路由的报文应该采用不同的规则，也可以为其单独创建一个 FIB 表......

为了支持多种路由算法和多个路由表，Linux 内核创造了一个名词：FIB 规则。这个规则是指处理报文时选取 FIB 表的规则。规则是策略性的关键性的新的概念。我们可以用自然语言这样描述规则，例如我门可以指定这样的规则：

规则一："所有来自 1.1.1.1 的 IP 包，使用路由表 253， 本规则的优先级别是 100"

规则二："所有的包，使用 252 号路由表，本规则的优先级别是 200"

优先级别越高的规则越先匹配（数值越小优先级别越高）。路由规则和 FIB 表之间的关系如下图所示：



图表 3-21 FIB 规则和 FIB 表的关系

如果没有配置 Policy Routing，我们也还是有路由规则的，但是，这种情况下的路由选择相对简单，就是本地和通向相邻主机的路由，于是在代码中就不需要 fib_rules 了（在 2.6.14 的代码中还是用到了这个变量），只是从概念上为这两种路由划分了"规则"，凡是本地路由，都使用 fib_local_table 这个数据库，而发往相邻主机的路由（目的地不一定是相邻主机），都使用 fib_main_table 这个数据库。

```
1.  struct fib_rule
2.  {
    FIB 规则实现成一个 r_next 的连接列表，r_clntref 时引用计数，它能够自动增加
3.  Struct fib_rule *r_next;
4.  Atomic_t    r_clntref;
     路由偏好值，这实际上类似于路由规则的优先级
5.  u32 r_preference;
     下面这个成员的名字不要引起误会，它实际更应该叫
     table_id，是指向由此规则选择的 FIB 表的索引，
     最大值为 255，这应该够用了
6.  uchar r_table;
     与 rtmsg 结构里的路由消息类型有相同的值
7.  uchar r_action;
8.  uchar r_dst_len;
9.  uchar r_src_len;
10. u32 r_src;//源地址
11. u32 r_srcmask;
12. u32 r_dst;//目的地址
13. u32 r_dstmask;
```

```
static struct fib_rule default_rule = {
    r_clntref: ATOMIC_INIT(2),
    r_preference: 0x7FFF,
    r_table: RT_TABLE_DEFAULT,
    r_action: RTN_UNICAST,
} ;
```

```
static struct fib_rule main_rule = {
    r_next: &default_rule,
    r_clntref: ATOMIC_INIT(2),
    r_preference: 0x7FFE,
    r_table: RT_TABLE_MAIN,
    r_action: RTN_UNICAST,
} ;
```

```
    r_srcmap 包含映射到源地址的地址，它用在有地址翻译的规则上
14. u32 r_srcmap;
15. u8 r_flags;
16. u8 r_tos;
    在此规则中使用的网络接口设备
17. int r_ifindex;
    网络接口设备的名字
18. char r_ifname[IFNAMSIZ];
19. int r_dead;
20. };
```

```
static struct fib_rule local_rule = {
    r_next: &main_rule,
    r_clntref: ATOMIC_INIT(2),
    r_table: RT_TABLE_LOCAL,
    r_action: RTN_UNICAST,
} ;
```

代码段 3-16 fib_rule 结构

有 3 个 fib_rule 实体去选择“永久的”FIB 表：main 表，local 表，如果没有指定的表，则选择缺省的表。当定义了一个新的 FIB 规则后，它会加到链表的末尾。

缺省规则表的 table 号是 253，指向 fib_tables 的第 253 号单元。依次类推，Main 表规则的号码是 254，Local 表规则的号码是 255，分别指向了 fib_tables 的相对单元。

如何进行进入、删除、查找规则呢？其实这是由应用程序使用 rtnetlink 接口来操作，首先通过从用户空间传递 nlmsghdr 消息到内核。nlmsghdr 结构放在 rtmsg 结构前面，包括了用来确定使用什么规则的大部分信息。内核 rtm 模块对这些消息进行解析，然后对应到合适的函数，如图中，当要增加一条路由规则时，就调用 inet_rtm_newrule，把新增的规则挂接到 fib_rules 链上，如果要增加一个 FIB 表，就调用 inet_rtm_newroute，并且参数中要指定创建一个新表，如此就把新增 FIB 表放入 fib_tables 中。由于这些不是我们的研究重点，所以我们可以跨过。

知道路由规则是干什么的了，那么我们可以下断言了：查找路由规则是查找路由的第一步！

是的，要查找路由则必须先搜索 fib_rules 以找到最匹配报文规则，必须同时满足 4 个条件：

1. 报文源地址与规则涵盖的源地址属于同一子网
2. 报文目的地址与规则涵盖的目的地址属于同一子网
3. 报文头设置的 TOS（如果配置了 tos 路由的话）与规则设置的一致
4. 报文始终的网络接口设备（如果指定了接口的话）与规则设置的一致

如果满足这 4 个条件，即搜索到一条这样的规则，找到了规则才能找 FIB 表。然后检查规则中的 r_action，如果我们为这样的 action 预备了一个 FIB 表，那么我们就得到正确的 FIB 表去搜索路由。对于缺省情况，r_action 都是 RTN_UNICAST，所以，local 表和 main 表是可以被搜索的。

（注：上面所说是 2.6.14 之前的代码运行原理，而 2.6.18 之后的代码已经不这么折腾了。在缺省情况下，不搜索 FIB 规则链表，直接去搜索 local 表和 main 表。我们会在下一章介绍）

找到了 FIB 表，我们就可以调用 FIB 表的自身的 lookup 函数，这类似于 C++中的概念：成员函数只能访问类本身的成员变量。现在搜索 FIB 表就是查找路由的第二步。

问题是 FIB 表本身不是由一个数据结构表示，而是由多个结构组合而成。前面说到我们缺省使用 HASH 算法进行查找，那么所谓的 hash 就在此刻出现了，而对于使用 LC-trie 算法的 FIB 表，我们不会看到 hash 这么一个数据结构。所以只能说，采用 hash 算法的 FIB 表系统是一个分层的结构组合。

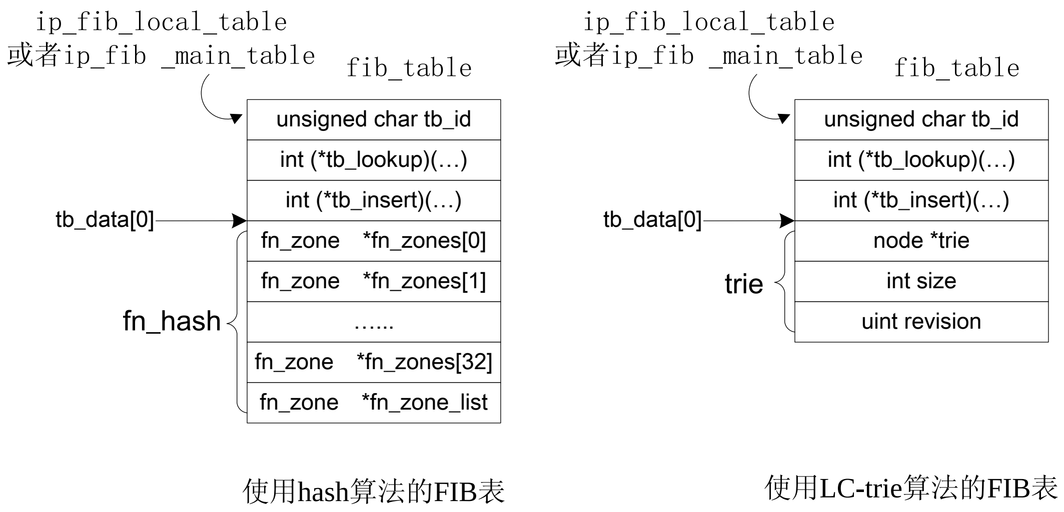
下面看 FIB 内部数据结构定义。

```
1.  struct fib_table
2.  {
    tb_id 字段是表的 id，如果配置了多个表，它的值在 1 到 255 之间。如果没有配置多个表，则只会
    RT_TABLE_MAIN 或 RT_TABLE_LOCAL，tb_stamp 目前没有使用。
        uchar tb_id;
        uint tb_stamp;
```

```
        int (*tb_lookup)(...);
        int (*tb_insert)(...);
        int (*tb_delete)(...);
    从表中获取所有的路由，这主要由 rtnetlink 使用 inet_dump_fib 调用它。
        int (*tb_dump)(...)
    操作删除 fib_table 中所有的表项
        int (*tb_flash)(...)
    选择缺省的路由
        void (*tb_select_default)
    tb_data 是指向 hash 表项的一个不透明指针，此表中的其他函数操作此字段，它不能被直接访问。
3.      uchar tb_data[0];
4.  }
```

代码段 3-17 fib_table 结构

最后一个数据成员是 tb_data[0]，它利用了编译器的一些特点，专门用来存放未知大小的数据。当用 hash 算法时，这个数据成员就是一个 fn_hash{}结构；而当使用 LC-trie 算法时，它却是一个 trie{}结构，这样可以充分节省空间。为什么不用 void*指针呢？因为在创建一个 fib_table 的同时，就已经为这些数据结构预留了空间，避免了再一次申请内存，操作起来也方便（请仔细观察上面的 fib_hash_init 第 10 行）。如下图：



图表 3-22 不同算法的 fib_table 的结构不同

Linux 的 FIB 表是分层的，从逻辑上来看可以分为 5 层：

第1层： 根据本地路由与否，分为 local FIB 库和 main FIB 库，这容易被大家忽略

第2层： FIB 表中的 fn_hash 数组由 33 个 fn_zone{}结构的指针组成。它将所有的路由根据子网掩码（netmask）的长度（0~32）分成 33 个部分（struct fn_zone），每一个 zones 代表一个唯一确定的 netmask。fn_hash{}最后一个成员指向掩码最长的 zone 区，这有什么用呢？我们在查找出口的那一章会告诉各位答案！

第3层： fn_zone{}也有一个 hash 表，凡子网掩码长度都相同的路由都放在其中，hash 表的节点叫 fib_node{}，比如 10.1.1.0/24 和 10.1.2.0/24 挂到了同一个 fn_zone 的链表上。

第4层： 在同一子网中，有可能由于 TOS 等属性的不同而使用不同的路由，那么每一个 fib_node{}节点有一个链表存放以 fib_alias{}结构为节点的路由表项，即使某 fib_node 上只有一条路由，也必须有一个 fib_alias{}保存路由的 tos、类型、范围等属性，不过但到此层还没有指出下一

跳出口。

第5层： fib_alias{}只包含一个 fib_info{}结构，此结构包括一些相应的参数，如协议，下一跳主机地址、外出的接口设备等等，这就是第 5 层。

分层的好处是显而易见的，它使路由表的更加优化，逻辑上也更加清淅，并且使数据可以共享（如 struct fib_info），从而减少了数据的冗余。下面我们一级一级的列出这几个结构的定义。

先上 fn_zone{}的结构，包含所有同长度子网掩码的路由：

```
1.  struct fn_zone {
    指向下一个 zone 的指针，比如目前是 fn_zone[24]，如果 fn_zone[16]有路由而 16 到 24 都没有路由，则指向 fn_zone[16]
2.      struct fn_zone      *fz_next;
    这个 hash 表结构才是体现这是 HASH 算法的地方。
3.      struct hlist_head   *fz_hash;
    表示在此 zone 的入口（实际就是 fib_node{}）的个数
4.      int          fz_nent;
    和此 zone 相关的 fz_hash 表的桶数，大多数 zones 都有 16 个，故此值为 16，但 0 zone 除外，它的值为 1
5.      int          fz_divisor;
    进入 fz_hash 的 mask，一般为 fz_divisor - 1
6.      u32          fz_hashmask;
7.     #define FZ_HASHMASK(fz)     ((fz)->fz_hashmask)
    表示该 zone 处于 fn_hash 表的什么位置
8.      int          fz_order;
    该 zone 所表示的掩码区，例如 16 位子网则是 0xFFFF，如果是 24 位，则是 0xFFFFFFF
9.      u32          fz_mask;
10.    #define FZ_MASK(fz)     ((fz)->fz_mask)
11. };
```

代码段 3-18 fn_zone 结构

每个路由的节点 fib_node{}，这个结构非常小，但是却是所谓 HASH 算法的来源，请注意那个 fn_key，它就是找到对应的 hash 节点的键值：

```
12. struct fib_node {
    链到 fz_hash 的节点，注意不是 fn_alias{}的 hash 链
13.     struct hlist_node   fn_hash;
    这个才是挂接 fn_alias{}的链表
14.     struct list_head    fn_alias;
    下面这个成员是最重要的值，实际上就是路由查找的关键，我们会在后面详述
15.     u32          fn_key;
16. };
```

代码段 3-19 fib_node 结构

如果有不同的 tos，那么节点可能有多个 fib_alias{}，不过其路由节点的属性其实是放在这个结构中，这里又看见一个“alias”单词，它不是指别名的意思，而是指到达同一路由的不同路径的集合（我个人的意见：有必要为了 tos 而专门设置一条链表吗？），要匹配这样一个 alias，要有唯一确定的 3 元组：<地址，tos，priority>，其中地址可以顺着 fib_node 找到，而后面两个是 fib_alias{}独有的，下一节我们还会看到这三元组：

```
1.  struct fib_alias {
    链到 fib_node→fn_alias 链的节点
2.  struct list_head    fa_list;
    关于此结点的更多的信息
3.  struct fib_info     *fa_info;
4.  u8          fa_tos;
   路由类型
5.  u8          fa_type;
```

表格 3-4

Route Type	Value	Route Types Explanation
RTN_UNSPEC	0	Route is unspecified. This is a gateway or direct route.
RTN_UNICAST	1	This route is a gateway or direct route.
RTN_LOCAL	2	在输入的时候接收这样的报文
RTN_BROADCAST	3	On input, accept packets locally as broadcast packets,and on output, send them as broadcast.
RTN_ANYCAST	4	Accept input packets locally as broadcast, but on output, send them as unicast packets.
RTN_MULTICAST	5	这是一条组播路由
RTN_BLACKHOLE	6	黑洞路由，丢掉这些报文
RTN_UNREACHABLE	7	目的不可达
RTN_PROHIBIT	8	管理员禁止了这条路由
RTN_THROW	9	此路由不在这个 FIB 库中
RTN_NAT	10	使用 NAT 协议转换这个地址
RTN_XRESOLVE	11	使用外部解析协议确定这条路由

```
    路由范围
6.  u8          fa_scope;
```

表格 3-5

宏	值	scope 描述
RT_SCOPE_UNIVERSE	0	当一个地址可以在任何地方使用时 scope 为 universe，当不为下面几种情况时就是该值
RT_SCOPE_SITE	200	在一个本地封闭系统中的内部路由
RT_SCOPE_LINK	253	当一个地址只在一个局域网内有意义且只在局域网内使用时，比如子网广播地址
RT_SCOPE_HOST	254	当一个地址只用于主机自身内部通信时 scope 为 host，该地址在主机以外不知道并且不能被使用，比如我们配置的 192.168.1.2 和 127.0.0.1 地址都是这种
RT_SCOPE_NOWHERE	255	目的地址不存在

```
    此路由的状态，目前只有 FA_S_ACCESSED 和 0 两个值，缺省是 0，如果被搜索过，则是前者。
7.  u8          fa_state;
8.  };
```

代码段 3-20 fib_alias 结构

在这个结构中我们接触到几个名词：

- Scope：在 Linux 中，路由的 scope 表示到目的网络距离的一种分类。
- Type：其实指得是主机如何处理路由，例如当发现此路由类型是 local 的，则应该上报，如果是 unicast，很可能转发。

路由类型和路由范围有一定关系。回忆之前给设备分配设备的时候，都是设置为 RTN_LOCAL，那么到达该 IP 地址的路由范围肯定是 RT_HOST，如果应用层路由协议比如 OSPF 往内核中设置一条到远端主机的路由，那么类型可以是 RTN_UNICAST，那么范围很可能是 RT_SCOPE_UNIVERSE。后面的章节将给出明证。

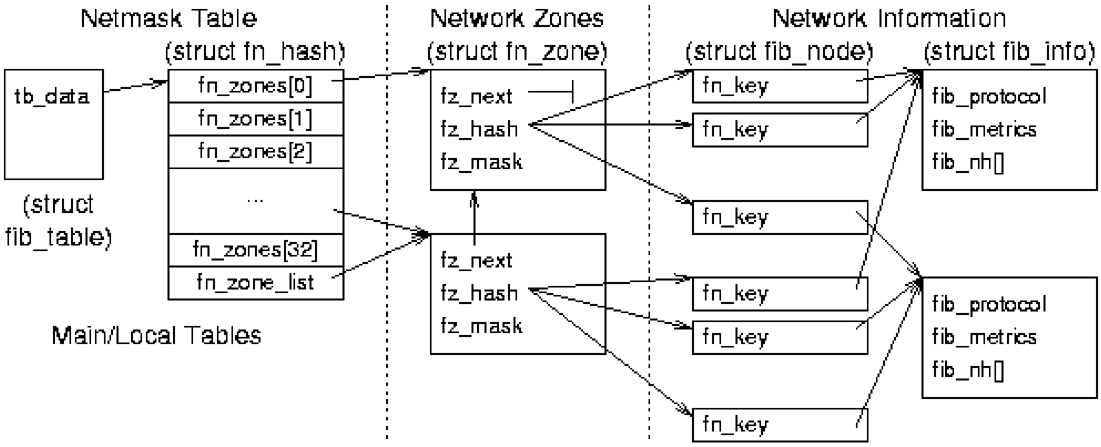
在 fib_semantics.c 这个文件中有一个 fib_props 结构数组，对我们了解两者之间的关系有一定帮助：

```
1.  static const struct
2.  {
3.  int error;
4.  u8  scope;
5.  } fib_props[RTA_MAX + 1] = {
6.      },  /* RTN_UNSPEC 类型的报文路由范围是 NOWHERE */
```

```
7.      {
8.          .error  = 0,
9.          .scope  = RT_SCOPE_UNIVERSE,
10.     },  /* RTN_UNICAST 类型的报文路由范围是 UNIVERSE*/
11.     {
12.         .error  = 0,
13.         .scope  = RT_SCOPE_HOST,
14.     },  /* RTN_LOCAL 类型的报文路由范围是 HOST */
15.     {
16.         .error  = 0,
17.         .scope  = RT_SCOPE_LINK,
18.     },  /* RTN_BROADCAST 类型的报文路由范围是 LINK*/
19.     {
20.         .error  = 0,
21.         .scope  = RT_SCOPE_LINK,
22.     },  /* RTN_ANYCAST 类型的报文路由范围是 LINK*/
23.     {
24.         .error  = 0,
25.         .scope  = RT_SCOPE_UNIVERSE,
26.     },  /* RTN_MULTICAST 类型的报文路由范围是 UNIVERSE*/
27.     {
28.         .error  = -EINVAL,
29.         .scope  = RT_SCOPE_UNIVERSE,
30.     },  /* RTN_BLACKHOLE */
31.     {
32.         .error  = -EHOSTUNREACH,
33.         .scope  = RT_SCOPE_UNIVERSE,
34.     },  /* RTN_UNREACHABLE */
35.     {
36.         .error  = -EACCES,
37.         .scope  = RT_SCOPE_UNIVERSE,
38.     },  /* RTN_PROHIBIT */
39.     {
40.         .error  = -EAGAIN,
41.         .scope  = RT_SCOPE_UNIVERSE,
42.     },  /* RTN_THROW */
43.     {
44.         .error  = -EINVAL,
45.         .scope  = RT_SCOPE_NOWHERE,
46.     },  /* RTN_NAT */
47.     {
48.         .error  = -EINVAL,
49.         .scope  = RT_SCOPE_NOWHERE,
50.     },  /* RTN_XRESOLVE */
51. };
```

代码段 3-21 fib_props 数组定义

fib_info{}包含关于一个接口的协议和特定的硬件信息，并且对于一些 zones 是公共信息，因为几个目的网络可以同时经过同一个接口出去。



图表 3-23fib_table 和 fn_zone、fib_node 结构的关系

```
1.  struct fib_info
2.  {
3.      struct fib_info *fib_next;
4.      struct fib_info *fib_prev;
5.      int fib_treeref;
6.      atomic_t fib_clntref;
7.      int fib_dead;
8.      unsigned fib_flags;
        指示本路由是哪个模块来创建的，其值就是 rtmsg→rtm_protocol，请参照前一节的介绍
9.      int fib_protocol;
        fib_prefsrc 是一个偏向使用的源地址，参照 RFC1122 文档中关于 UDP multihoming 部分，它作
    为一个“指定目的地址”，被 UDP 用作回应报文的源地址。
10.     u32 fib_prefsrc;
11.     u32 fib_priority;
12.     unsigned fib_metrics[RTAX_MAX];
13. #define fib_mtu fib_metrics[RTAX_MTU-1]
14. #define fib_window fib_metrics[RTAX_WINDOW-1]
15. #define fib_rtt fib_metrics[RTAX_RTT-1]
16. #define fib_advmss fib_metrics[RTAX_ADVMSS-1]
        fib_nhs 是下一跳的数量，如果该路由有一个已定义的网关，则应该是 1；否则它就是 0。但是，如果
    配置了多路径路由，则它有可能大于 1。
17.     int fib_nhs;
        fib_nh 包含关于下一跳或下一跳链表的信息，如果超过一个的话。fib_dev 是特指到达下一跳主机
    的网络接口设备。
18.     struct fib_nh fib_nh[0];
19. #define fib_dev fib_nh[0].nh_dev
20. }
```

代码段 3-22 fib_info 结构

顺便也介绍 fib_nh 数据结构，通过路由机制去抽取关于下一跳或网关的信息去修改它。

```
1.  struct fib_nh
2.  {
3.      struct net_device *nh_dev;
4.      uint nh_flags;
        nh_scope 和 fib_alias 结构里的 scope 字段相似，它并没有真正定义路由的范围，而是概念上到
    目的主机的距离，在这种情况下，是网关。
5.      uchar nh_scope;
        输出到网关机器接口的索引号
6.      int nh_oif;
```

```
7.  }
```

代码段 3-23 fib_nh 结构

这一节完整的讲述了 FIB 表的组成结构，可能读者们一下子还没有明白是什么意思，那我们先继续配置的例子，把 FIB 的故事继续讲完。给大家一个感性的认识。

3.4 FIB 系统发生了什么样的变化

上次说到当给设备设置 IP 地址的的时候，内核给 inetdev_chain 通知链发送了一个 NETDEV_UP 事件，FIB 系统正好对这个事件感兴趣，就把下面这个结构注册到了 inetaddr_chain 上：

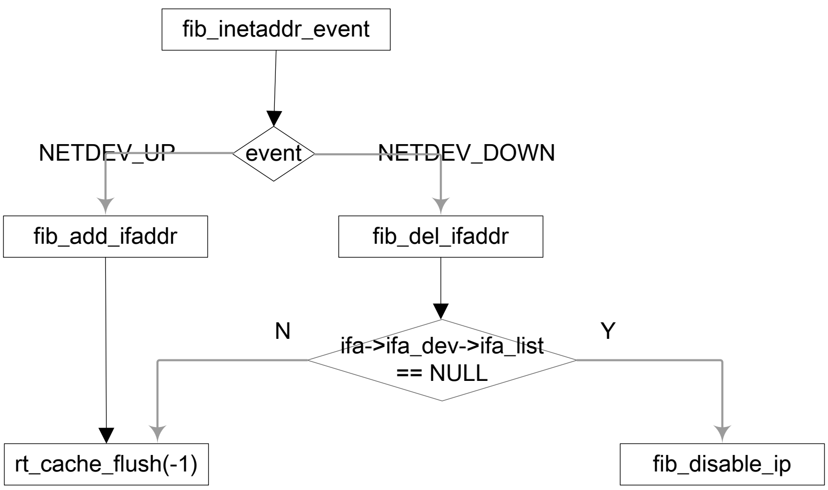
static struct notifier_block fib_inetaddr_notifier = {

.notifier_call = fib_inetaddr_event,

};

为什么 FIB 系统会对这个事情感兴趣？因为给设备分配 IP 地址相对于给系统增加一条 type 为 RTN_LOCAL 的路由，其路由范围是 RT_SCOPE_HOST，没错吧？给系统增加 IP 地址，其实也就和路由软件（OSPF、RIP 等）往内核里增加路由表项一样，基本的流程相同，只是个别参数不同而已（我甚至觉得，路由可以理解为地址，区别在于这是别人的地址☺，以后如果实在不理解路由的本质，你就把它认为是别人的地址就得了）。重点在这里是你了解了 FIB 系统是如何增删路由表项的。

其回调函数就是 fib_inetaddr_event，此函数如果收到 NETDEV_DOWN 消息，它就删除 FIB 中存在的地址，如果 ifa_dev→ifa_list 是空，则 disable 这个设备，否则就刷新一下路由 cache（注意，这里是路由 cache，不是 FIB 表）：



图表 3-24fib_inetaddr_event 函数内部实现

很明显，目前的流程是走向了左边。其参数就是 inet_insert_ifa 中传入的 ifa。我们只是访问其成员变量，而不再更改。

```
1.
2.  void fib_add_ifaddr(struct in_ifaddr *ifa)
3.  {
4.      struct in_device *in_dev = ifa->ifa_dev;
5.      struct net_device *dev = in_dev->dev;
```

```
6.      struct in_ifaddr *prim = ifa;
7.      u32 mask = ifa->ifa_mask;
8.      u32 addr = ifa->ifa_local;
9.      u32 prefix = ifa->ifa_address&mask;
10.
11.     if (ifa->ifa_flags&IFA_F_SECONDARY) {
12.         prim = inet_ifa_byprefix(in_dev, prefix, mask);
13.         ......
14.     }
        如果是 loopback 接口配置，addr 是 127.0.0.1，而配置例子中 192.168.18.2
15.     ①fib_magic(RTM_NEWROUTE, RTN_LOCAL, addr, 32, prim);
16.
17.     if (!(dev->flags&IFF_UP))
18.         return;
19.
20.     /* Add broadcast address, if it is explicitly assigned. */
21.     if (ifa->ifa_broadcast && ifa->ifa_broadcast != 0xFFFFFFFF)
         ifa->ifa_broadcast 是 192.168.18.255，而对于 loopback 接口是不会进入这一行
22.         ②fib_magic(RTM_NEWROUTE,RTN_BROADCAST,ifa->ifa_broadcast,32,prim);
23.
24.     if (!ZERONET(prefix) && !(ifa->ifa_flags&IFA_F_SECONDARY) &&
25.         (prefix != addr || ifa->ifa_prefixlen < 32)) {
         第二个参数 loopback 接口是 RTN_LOCAL,prefix 是 127,而配置例子对应的是 RTN_UNICAST，
         prefix 是 192.168.18，这导致它们存取的 FIB 表不一样。
26.         ③fib_magic(RTM_NEWROUTE, dev->flags&IFF_LOOPBACK ? RTN_LOCAL :
27.                 RTN_UNICAST, prefix, ifa->ifa_prefixlen, prim);
28.
29.         /* Add network specific broadcasts, when it takes a sense */
30.         if (ifa->ifa_prefixlen < 31) {
        loopback 接口 prefix 是 127, 而配置例子是 192.168.18
31.             ④fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix, 32, prim);
        loopback 接口的 prefix|~mask 是 255.255.255.127，而配置例子是 192.168.18.255
32.             ⑤fib_magic(RTM_NEWROUTE, RTN_BROADCAST, prefix|~mask, 32, prim);
33.         }
34.     }
35. }
```

代码段 3-24 fib_add_ifaddr 函数

我们已经给出函数中调用 fib_magic 的第三个参数。根据这些参数我们看看它的执行是什么结果：

```
    创建并初始化一个核内路由命令消息，这实际是和 netlink 消息的处理过程（请参考 inet_rtm_newaddr
    函数）一样，但是又不能直接引用 netlink 的代码，因为不太好设置传入的参数。在处理这些 FIB 引擎时，
    netlink 已经被锁住
1.  static void fib_magic(int cmd, int type, u32 dst, int dst_len, struct in_ifaddr
    *ifa)
2.  {
3.      struct fib_table * tb;
4.      struct {
5.          struct nlmsghdr nlh;
6.          struct rtmsg    rtm;
7.      } req;
8.      struct kern_rta rta;
9.
10.     memset(&req.rtm, 0, sizeof(req.rtm));
11.     memset(&rta, 0, sizeof(rta));
        对于第 ③ 次进入，才会得到 MAIN 表，其余几次都是 LOCAL 表，在没有定义
        CONFIG_IP_MULTIPLE_TABLES 的情况下，其实就是获得 ip_fib_main_table 或
        ip_fib_local_table的指针，而不是根据其函数名创建一个新的FIB表。对于loopback接口类地
        址，它访问的都是local表，不可能访问main表
12.     if (type == RTN_UNICAST)
```

```
13.         tb = fib_new_table(RT_TABLE_MAIN);
14.     else
15.         tb = fib_new_table(RT_TABLE_LOCAL);
     下面就开始构造一个 nlm 和 rtm 消息体，其实仅仅是把它们传入 tb_insert 函数，而非要把他们真正
     发送出去
16.     req.nlh.nlmsg_len = sizeof(req);
17.     req.nlh.nlmsg_type = cmd;
18.     req.nlh.nlmsg_flags = NLM_F_REQUEST|NLM_F_CREATE|NLM_F_APPEND;
19.     req.nlh.nlmsg_pid = 0;
20.     req.nlh.nlmsg_seq = 0;
21.
22.     req.rtm.rtm_dst_len = dst_len;
23.     req.rtm.rtm_table = tb->tb_id;
        要记住这种方式下尽管是用户运行 ifconfig 命令产生了路由，但实际是由内核产生了路由
24.     req.rtm.rtm_protocol = RTPROT_KERNEL;
        只有第①次传入 127.0.0.1 或 192.168.18.2 的时候rtm_scope的值是HOST，其余都是LINK
25.     req.rtm.rtm_scope = (type != RTN_LOCAL ? RT_SCOPE_LINK : RT_SCOPE_HOST);
26.     req.rtm.rtm_type = type;
27.
28.     rta.rta_dst = &dst;
29.     rta.rta_prefsrc = &ifa->ifa_local;
       这个步骤很重要，是后面查询 FIB 时找到出接口的关键
30.     rta.rta_oif = &ifa->ifa_dev->dev->ifindex;
        往系统配置地址本质就是往路由系统增加一个比较特殊的路由，所以 cmd 是 NEWROUTE，而不是
        NEWADDR。
31.     if (cmd == RTM_NEWROUTE)
32.         tb->tb_insert(tb, &req.rtm, &rta, &req.nlh, NULL);
33.     else
34.         tb->tb_delete(tb, &req.rtm, &rta, &req.nlh, NULL);
35. }
```

代码段 3-25　fib_magic 函数

tb->tb_insert 在此指向了 fn_hash_insert。

```
1.  static int
2.  fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
3.        struct nlmsghdr *n, struct netlink_skb_parms *req)
4.  {
5.      struct fn_hash *table = (struct fn_hash *) tb->tb_data;
6.      struct fib_node *new_f, *f;
7.      struct fib_alias *fa, *new_fa;
8.      struct fn_zone *fz;
9.      struct fib_info *fi;
        上面的代码中 rtm_dst_len 的长度等于 32
10.     int z = r->rtm_dst_len;
11.     int type = r->rtm_type;
12.     u8 tos = r->rtm_tos;
13.     u32 key;
14.     int err;
15.
16.     if (z > 32)
17.         return -EINVAL;
        z的值如下：对于 192.168.18.1/24，第①、②次进入是 32，第③次进入是 24，第④、⑤
        进入是 32
        对于 127.0.0.1，第①次进入是 32，而第②次不会被执行，第③次进入是 8，第④、⑤进入
        是 32
18.     fz = table->fn_zones[z];
19.     if (!fz && !(fz = fn_new_zone(table, z)))
20.         return -ENOBUFS;
21.
22.     key = 0;
23.     if (rta->rta_dst) {
24.         u32 dst;
25.         memcpy(&dst, rta->rta_dst, 4);
```

```
            保证目的地址? ? ? ? ? ,
26.         if (dst & ~FZ_MASK(fz))
27.             return -EINVAL;
           获取路由表项的键值，这是最重要的一个步骤。计算方式就是 dst&fz→fz_mask 的值。原来
           所谓键值就是一个地址
28.         key = fz_key(dst, fz);
29.     }
30.
31.     fi = fib_create_info(r, rta, n, &err);
32.
33.     if (fz->fz_nent > (fz->fz_divisor<<1) &&
34.         fz->fz_divisor < FZ_MAX_DIVISOR &&
35.         (z==32 || (1<<z) > fz->fz_divisor))
36.         fn_rehash_zone(fz);
        通过 key 查找是否曾经创建过 fib_info{}结构，毕竟一个唯一的路由/地址只能有一个 key
37.     f = fib_find_node(fz, key);
38.
39.     if (!f)
            如果之前没有创建 fib_node{}，那么肯定就没有创建过相关的 fib_alias{}
40.         fa = NULL;
41.     else
            如果有 fib_node{},那么肯定有 fib_alias{}结构，但只是想找出和 tos 及优先级都相同
            的 fib_alias{}，还不一定有。不过在我们目前的状况下，肯定能找到一个。
42.         fa = fib_find_alias(&f->fn_alias, tos, fi->fib_priority);
        有 3 种情况：
        1)如果没有没有找到 tos 及优先级相同的 fib_alias{}，那就创建一个插入并到 fib_node{}的
        前面

        2)如果连与地址相配的 fib_node{}都没有找到，我们就要创建一个
43.
44.     if (fa && fa->fa_tos == tos && fa->fa_info->fib_priority == fi->fib_priority)
45.     {
        3)如果找到 fib_alias{}，而其 3 元组<地址，tos，priority>却和我们的一致，那就不要再
        创建了 fib_alias{}，甚至连 fib_info{}也不要创建了。
46.     struct fib_alias *fa_orig;
47.
48.     err = -EEXIST;
49.     if (n->nlmsg_flags & NLM_F_EXCL)
50.         goto out;
51.
52.     if (n->nlmsg_flags & NLM_F_REPLACE) {
        如果是要代替这个路由表项，就把第 31 行创建 fib_info{}赋给 fib_alias{},原来老的删除掉
53.         struct fib_info *fi_drop;
54.         u8 state;
55.
56.         fi_drop = fa->fa_info;
57.         fa->fa_info = fi;
58.         fa->fa_type = type;
59.         fa->fa_scope = r->rtm_scope;
60.         state = fa->fa_state;
61.         fa->fa_state &= ~FA_S_ACCESSED;
62.         fib_hash_genid++;
63.
64.         fib_release_info(fi_drop);
65.         if (state & FA_S_ACCESSED)
66.             rt_cache_flush(-1);
67.         return 0;
68.     }
69.
70.     /* Error if we find a perfect match which
71.      * uses the same scope, type, and nexthop
72.      * information.
73.      */
74.     fa_orig = fa;
75.     fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
76.     list_for_each_entry_continue(fa, &f->fn_alias, fa_list) {
77.         if (fa->fa_tos != tos)
```

```
78.             break;
79.         if (fa->fa_info->fib_priority != fi->fib_priority)
80.             break;
81.         if (fa->fa_type == type &&
82.             fa->fa_scope == r->rtm_scope &&
83.             fa->fa_info == fi)
84.             goto out;
85.      }
86.      if (!(n->nlmsg_flags & NLM_F_APPEND))
87.     fa = fa_orig;
88. }
89.
90. err = -ENOBUFS;
    创建 fib_alias{}结构
91. new_fa = kmem_cache_alloc(fn_alias_kmem, SLAB_KERNEL);
92.
93. new_f = NULL;
94. if (!f)
95. {
        创建 fib_node{}结构
96.     new_f = kmem_cache_alloc(fn_hash_kmem, SLAB_KERNEL);
97.
98.     INIT_HLIST_NODE(&new_f->fn_hash);
99.     INIT_LIST_HEAD(&new_f->fn_alias);
        把键值赋给这个 fib_node{}
100.    new_f->fn_key = key;
101.    f = new_f;
102.    }
103.
104.    new_fa->fa_info = fi;
105.    new_fa->fa_tos = tos;
106.    new_fa->fa_type = type;
107.    new_fa->fa_scope = r->rtm_scope;
108.    new_fa->fa_state = 0;
109.
110.    /*
111.     * Insert new entry to the list.
112.     */
113.
114.    if (new_f)
115.        fib_insert_node(fz, new_f);
116.    list_add_tail(&new_fa->fa_list,
117.         (fa ? &fa->fa_list : &f->fn_alias));
118.    fib_hash_genid++;
119.
120.    if (new_f)
121.        fz->fz_nent++;
122.    rt_cache_flush(-1);
123.
124.    rtmsg_fib(RTM_NEWROUTE, key, new_fa, z, tb->tb_id, n, req);
125.        return 0;
126.
127.    out_free_new_fa:
128.        kmem_cache_free(fn_alias_kmem, new_fa);
129.    out:
130.        fib_release_info(fi);
131.        return err;
132.    }
```

代码段 3-26 fn_hash_insert 函数

当对应掩码长度的 zone 没有被创建时，就应该创建一个。里面的成员基本上都能根据掩码的长度 z 来确定。

```
1.  static struct fn_zone *
2.  fn_new_zone(struct fn_hash *table, int z)
3.  {
4.      int i;
```

```
5.      struct fn_zone *fz = kzalloc(sizeof(struct fn_zone), GFP_KERNEL);
    对于目前的情况，z 要么是 32，要么是 24，所以 fz_divisor 是 16
6.      if (z) {
7.          fz->fz_divisor = 16;
8.      } else {
9.          fz->fz_divisor = 1;
10.     }
11.     fz->fz_hashmask = (fz->fz_divisor - 1);
     根据第 6 行，fz_hash 数组有 16 个单元
12.     fz->fz_hash = fz_hash_alloc(fz->fz_divisor);
13.
14.     memset(fz->fz_hash, 0, fz->fz_divisor * sizeof(struct hlist_head *));
15.     fz->fz_order = z;
      要注意这个制造 mask 的函数把掩码转成主机字节序
16.     fz->fz_mask = inet_make_mask(z);
17.
      找到第一个非空的 zone，并且该 zone 的掩码比当前的更长
18.     for (i=z+1; i<=32; i++)
19.         if (table->fn_zones[i])
20.             break;
     如果 i 大于 32，说明当前 zone 具有当前
     系统中最精确的掩码
21.     if (i>32) {
        这里有 3 种可能：
        1)当前 zone 就是 FIB 系统里第一个成员，因为遍历了整个 zone，发现都是空的（z 是 0）
        2)当前 zone 的掩码比以前的 zone 的掩码都要长，因为 i 从 z＋1 开始。
        3)当前 zone 就是 32 位掩码长度，根本不需要遍历 fn_zones[]了。
22.         fz->fz_next = table->fn_zone_list;
        fn_zone_list 指针一定要指向最长掩码的 zone
23.         table->fn_zone_list = fz;
24.     } else {
          把此 zone 插入到最精确 zone 和次精确 zone 之间
25.         fz->fz_next = table->fn_zones[i]->fz_next;
26.         table->fn_zones[i]->fz_next = fz;
27.     }
       终于把创建的 zone 的指针赋给了数组
28.     table->fn_zones[z] = fz;
29.     fib_hash_genid++;
30.
31.     return fz;
32. }
```

```
__u32 inet_make_mask(int logmask)
{
    if (logmask)
        return htonl(~((1<<(32-logmask))-1));
    return 0;
}
```

代码段 3-27　fn_new_zone 函数

按上节说的 FIB 库分层结构，创建了 zone 就应该创建 fib_node{}了，可是没有，它先创建了 fib_info{}，先硬着头皮往下看吧。

```
1. struct fib_info *
2. fib_create_info(const struct rtmsg *r, struct kern_rta *rta,
3.  const struct nlmsghdr *nlh, int *errp)
4. {
5.      int err;
6.      struct fib_info *fi = NULL;
7.      struct fib_info *ofi;
       如果是配置支持路由策略，那么 nhs（下一跳接口的个数）有可能大于 1，但是这种情况我们不考虑。
8.      #ifdef CONFIG_IP_ROUTE_MULTIPATH
9.      int nhs = 1;
10.     #else
       缺省情况下，nhs 只能等于 1，所有用 cost 约束
11.     const int nhs = 1;
12.     #endif
13.
14.     /* Fast check to catch the most weird cases */
      下面这种情况出现的可能性非常小，当配置 127.0.0.1 或当配置 192.168.1.1 时 r→rtm_type 是 2，
      而 r→rtm_scope 是 254
15.     if (fib_props[r->rtm_type].scope > r->rtm_scope)
```

```
16.         goto err_inval;
17.
18.     #ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
19.     if (rta->rta_mp_alg) {
20.         mp_alg = *rta->rta_mp_alg;
21.
22.         if (mp_alg < IP_MP_ALG_NONE ||
23.            mp_alg > IP_MP_ALG_MAX)
24.             goto err_inval;
25.     }
26.     #endif
27.
28.     err = -ENOBUFS;
    下面这 2 个全局变量在初始化的时候都等于 0，所以肯定能在配置第一个地址的时候进入这个判断
29.     if (fib_info_cnt >= fib_hash_size)
30.     {
      fib_hash_size 的值每次都增大 1 倍，即 1→2→4→8→......，但是它是如何从 0 变成 1 呢？
31.         unsigned int new_size = fib_hash_size << 1;
32.         struct hlist_head *new_info_hash;
33.         struct hlist_head *new_laddrhash;
34.         unsigned int bytes;
    就在这里，由于 new_size 变成 1，并在 fib_hash_move 函数中赋给了 fib_hash_size，使之完
    成 0→1 的变化
35.         if (!new_size)
36.             new_size = 1;
    根据 new_size 重新创建 2 个 hash 表数组,这 2 个 hash 表就是全局的 hash 表,会在 fib_hash_move
    函数中分别赋给 fib_info_hash 和 fib_info_laddrhash。
37.         bytes = new_size * sizeof(struct hlist_head *);
38.         new_info_hash = fib_hash_alloc(bytes);
39.         new_laddrhash = fib_hash_alloc(bytes);
40.         ......
41.         {
42.             memset(new_info_hash, 0, bytes);
43.             memset(new_laddrhash, 0, bytes);
    不要被它的名字迷惑了,它内部确实是一个一个地清除了 fib_info_hash 和 fib_info_laddrhash
    里的节点，并释放了这 2 个表数组，但是，它又将原来的节点都保存到 new_info_hash 和
    new_laddrhash 中了（实际就是那 2 个全局表）
44.             fib_hash_move(new_info_hash, new_laddrhash, new_size);
45.         }
    下面这种情况几乎不会发生，除非 fib_hash_alloc 失败，所以请无视之
46.         if (!fib_hash_size)
47.             goto failure;
48.     }
    在没有配置成多路径的情况下，nhs 是 1，那么 fib_info 中的 fib_nh 数组就只有一个单元
49.     fi = kzalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);
50.
51.     fib_info_cnt++;
    在这里是 RTPROT_KERNEL，不是我们常说的 TCP 或 UDP，如果是某个路由协议比如 OSPF，则是那个
    “协议”自己指定的值，比如 RTPROT_ZEBRA（zebra 是一个比较大的路由软件开发公司）
52.     fi->fib_protocol = r->rtm_protocol;
53.
54.     fi->fib_nhs = nhs;
55.     change_nexthops(fi) {
56.         nh->nh_parent = fi;
57.     } endfor_nexthops(fi)
58.
59.     fi->fib_flags = r->rtm_flags;
60.     if (rta->rta_priority)
61.         fi->fib_priority = *rta->rta_priority;
62.     if (rta->rta_mx) {
63.         int attrlen = RTA_PAYLOAD(rta->rta_mx);
64.         struct rtattr *attr = RTA_DATA(rta->rta_mx);
65.
66.         while (RTA_OK(attr, attrlen)) {
67.             unsigned flavor = attr->rta_type;
68.             if (flavor) {
69.                 if (flavor > RTAX_MAX)
```

```
70.                     goto err_inval;
71.                 fi->fib_metrics[flavor-1] = *(unsigned*)RTA_DATA(attr);
72.             }
73.             attr = RTA_NEXT(attr, attrlen);
74.         }
75.     }
76.     if (rta->rta_prefsrc)
77.         memcpy(&fi->fib_prefsrc, rta->rta_prefsrc, 4);
78.
79.     if (rta->rta_mp) {
        如果是支持策略路由的话就进入下面的分支
80.     #ifdef CONFIG_IP_ROUTE_MULTIPATH
81.         if ((err = fib_get_nhs(fi, rta->rta_mp, r)) != 0)
82.             goto failure;
83.         if (rta->rta_oif && fi->fib_nh->nh_oif != *rta->rta_oif)
84.             goto err_inval;
85.         if (rta->rta_gw && memcmp(&fi->fib_nh->nh_gw, rta->rta_gw, 4))
86.             goto err_inval;
87.     #ifdef CONFIG_NET_CLS_ROUTE
88.         if (rta->rta_flow && memcmp(&fi->fib_nh->nh_tclassid, rta->rta_flow, 4))
89.             goto err_inval;
90.     #endif
91.     #else
92.         goto err_inval;
93.     #endif
94.     } else {
95.         struct fib_nh *nh = fi->fib_nh;
            我们曾经指定了 rta_oif
96.         if (rta->rta_oif)
97.             nh->nh_oif = *rta->rta_oif;
            我们目前没有设置网关，所以 rta_gw 是 NULL
98.         if (rta->rta_gw)
99.             memcpy(&nh->nh_gw, rta->rta_gw, 4);
100.    #ifdef CONFIG_NET_CLS_ROUTE
101.        if (rta->rta_flow)
102.            memcpy(&nh->nh_tclassid, rta->rta_flow, 4);
103.    #endif
104.        nh->nh_flags = r->rtm_flags;
105.    #ifdef CONFIG_IP_ROUTE_MULTIPATH
106.        nh->nh_weight = 1;
107.    #endif
108.    }
109.
110.    if (fib_props[r->rtm_type].error) {
111.        if (rta->rta_gw || rta->rta_oif || rta->rta_mp)
112.            goto err_inval;
113.        goto link_it;
114.    }
115.
116.    if (r->rtm_scope > RT_SCOPE_HOST)
117.        goto err_inval;
         我们配置主机地址时，scope 是 HOST，
118.    if (r->rtm_scope == RT_SCOPE_HOST) {
         下面指定下一跳设备就是该地址的 owner（设备接口），对于参数为 127.0.0.1 或 192.168.18.2，
          则是进入此分支
119.        struct fib_nh *nh = fi->fib_nh;
120.
121.        /* Local address is added. */
122.        if (nhs != 1 || nh->nh_gw)
123.            goto err_inval;
124.        nh->nh_scope = RT_SCOPE_NOWHERE;
            根据上面分配的接口索引找到对应 net_device{}
125.        nh->nh_dev = dev_get_by_index(fi->fib_nh->nh_oif);
126.        ......
127.    } else {
            如果是远端主机或者某局域网地址，就要遍历 fi 的所有下一跳，但对于缺省情况，只循环一次。
            fib_check_nh 完成比较复杂的事情，见下面的函数
128.        change_nexthops(fi) {
```

```
            在这里 bh 是 fi->fib_nh 成员
129.            if ((err = fib_check_nh(r, fi, nh)) != 0)
130.                goto failure;
131.        } endfor_nexthops(fi)
132.    }
133.
134.    if (fi->fib_prefsrc) {
135.        if (r->rtm_type != RTN_LOCAL || rta->rta_dst == NULL ||
136.            memcmp(&fi->fib_prefsrc, rta->rta_dst, 4))
137.            if (inet_addr_type(fi->fib_prefsrc) != RTN_LOCAL)
138.                goto err_inval;
139.    }
140.
141.    link_it:
        查找和刚才创建的 fib_info 信息相同的节点，如果在 hash 表中找到了，那么就释放该 fi，并把
        fib_treeref 自增，表示该节点的又被引用了一次
142.    if ((ofi = fib_find_info(fi)) != NULL) {
143.        fi->fib_dead = 1;
144.        free_fib_info(fi);
145.        ofi->fib_treeref++;
146.        return ofi;
147.    }
148.
149.    fi->fib_treeref++;
150.
        把 62 行创建的 fib_info 放入 fib_info_hash 表中，如果 key 冲突就挂到 hash 链前面
151.    hlist_add_head(&fi->fib_hash,
152.              &fib_info_hash[fib_info_hashfn(fi)]);
153.    if (fi->fib_prefsrc) {
154.        struct hlist_head *head;
155.
156.        head = &fib_info_laddrhash[fib_laddr_hashfn(fi->fib_prefsrc)];
157.        hlist_add_head(&fi->fib_lhash, head);
158.    }
159.    change_nexthops(fi) {
160.        struct hlist_head *head;
161.        unsigned int hash;
162.
163.        if (!nh->nh_dev)
164.            continue;
165.        hash = fib_devindex_hashfn(nh->nh_dev->ifindex);
166.        head = &fib_info_devhash[hash];
      把 fib_nh 放入 fib_info_devhash 表中，如果 key 冲突就挂到前面
167.        hlist_add_head(&nh->nh_hash, head);
168.    } endfor_nexthops(fi)
169.
170.    return fi;
171.
172.    err_inval:
173.    err = -EINVAL;
174.
175.    failure:
176.           *errp = err;
177.           if (fi) {
178.        fi->fib_dead = 1;
179.        free_fib_info(fi);
180.    }
181.    return NULL;
182.    }
```

代码段 3-28 fib_create_info 函数

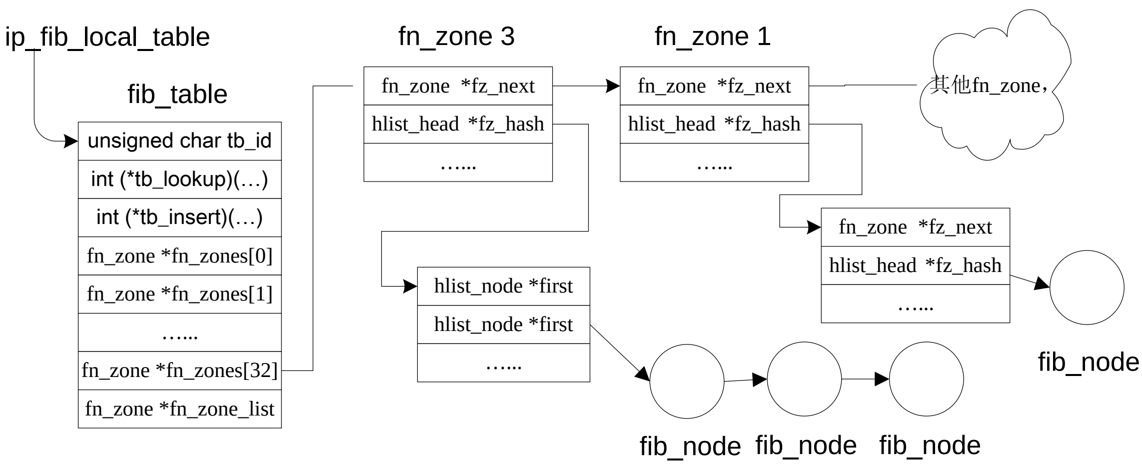
从这份代码可以推断出 fib_info 和 fib_nh 的关系，要注意的是 fib_nh 里的 nh_dev 是根据自身的 nh_oif 查找到的，这是 fib_magic 函数里把 ifa->ifa_dev->dev->ifindex 赋给 rta.rta_oif，然后传到这里来的。

```
1.  /*
```

```
2.    Semantics of nexthop is very messy by historical reasons.
3.    We have to take into account, that:
4.    a) gateway can be actually local interface address,
5.       so that gatewayed route is direct.
6.    b) gateway must be on-link address, possibly
7.       described not by an ifaddr, but also by a direct route.
8.    c) If both gateway and interface are specified, they should not
9.       contradict.
10.   d) If we use tunnel routes, gateway could be not on-link.
11.
12.   Attempt to reconcile all of these (alas, self-contradictory) conditions
13.   results in pretty ugly and hairy code with obscure logic.
14.
15.   I chose to generalized it instead, so that the size
16.   of code does not increase practically, but it becomes
17.   much more general.
18.   Every prefix is assigned a "scope" value: "host" is local address,
19.   "link" is direct route,
20.   [ ... "site" ... "interior" ... ]
21.   and "universe" is true gateway route with global meaning.
22.
23.   Every prefix refers to a set of "nexthop"s (gw, oif),
24.   where gw must have narrower scope. This recursion stops
25.   when gw has LOCAL scope or if "nexthop" is declared ONLINK,
26.   which means that gw is forced to be on link.
27.
28.   Code is still hairy, but now it is apparently logically
29.   consistent and very flexible. F.e. as by-product it allows
30.   to co-exists in peace independent exterior and interior
31.   routing processes.
32.
33.   Normally it looks as following.
34.
35.   {universe prefix}  -> (gw, oif) [scope link]
36.                        |
37.          |-> {link prefix} -> (gw, oif) [scope local]
38.                              |
39.                    |-> {local prefix} (terminal node)
40.  */
41.
42. static int fib_check_nh(const struct rtmsg *r, struct fib_info *fi, struct fib_nh
    *nh)
43. {
44. int err;
45. if (nh->nh_gw) {
    如果指定了网关地址，就要
46.
47.     struct fib_result res;
48.
49. #ifdef CONFIG_IP_ROUTE_PERVASIVE
50.     if (nh->nh_flags&RTNH_F_PERVASIVE)
51.         return 0;
52. #endif
53.     if (nh->nh_flags&RTNH_F_ONLINK) {
54.         struct net_device *dev;
55.
56.         if (r->rtm_scope >= RT_SCOPE_LINK)
57.             return -EINVAL;
58.         if (inet_addr_type(nh->nh_gw) != RTN_UNICAST)
59.             return -EINVAL;
60.         if ((dev = __dev_get_by_index(nh->nh_oif)) == NULL)
61.             return -ENODEV;
62.         if (!(dev->flags&IFF_UP))
63.             return -ENETDOWN;
64.         nh->nh_dev = dev;
65.         nh->nh_scope = RT_SCOPE_LINK;
66.         return 0;
67.     }
68.     {
```

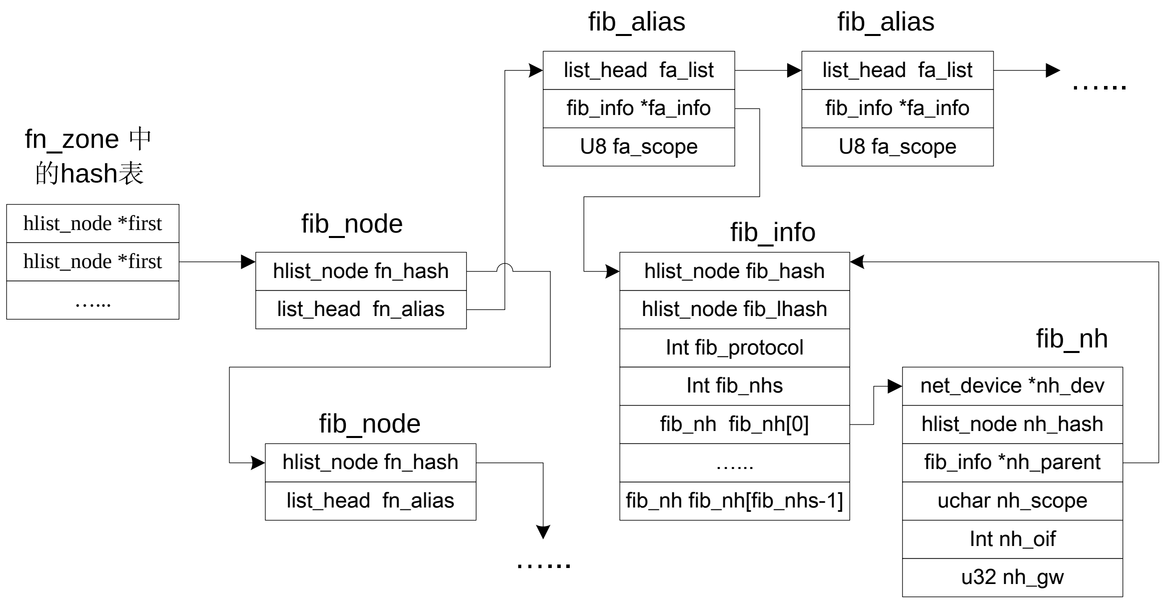
```
69.         struct flowi fl = { .nl_u = { .ip4_u =
70.                         { .daddr = nh->nh_gw,
71.                         .scope = r->rtm_scope + 1 } },
72.                     .oif = nh->nh_oif };
73.
74.         /* It is not necessary, but requires a bit of thinking */
75.         if (fl.fl4_scope < RT_SCOPE_LINK)
76.             fl.fl4_scope = RT_SCOPE_LINK;
77.         if ((err = fib_lookup(&fl, &res)) != 0)
78.             return err;
79.     }
80.     err = -EINVAL;
81.     if (res.type != RTN_UNICAST && res.type != RTN_LOCAL)
82.         goto out;
83.     nh->nh_scope = res.scope;
84.     nh->nh_oif = FIB_RES_OIF(res);
85.     if ((nh->nh_dev = FIB_RES_DEV(res)) == NULL)
86.         goto out;
87.     err = -ENETDOWN;
88.     if (!(nh->nh_dev->flags & IFF_UP))
89.         goto out;
90.     err = 0;
91. out:
92.     fib_res_put(&res);
93.     return err;
94. } else {
95.     struct in_device *in_dev;
96.
97.     if (nh->nh_flags&(RTNH_F_PERVASIVE|RTNH_F_ONLINK))
98.         return -EINVAL;
99.
100.            in_dev = inetdev_by_index(nh->nh_oif);
101.            if (in_dev == NULL)
102.                return -ENODEV;
103.            if (!(in_dev->dev->flags&IFF_UP)) {
104.                in_dev_put(in_dev);
105.                return -ENETDOWN;
106.            }
107.            nh->nh_dev = in_dev->dev;
108.            dev_hold(nh->nh_dev);
109.            nh->nh_scope = RT_SCOPE_HOST;
110.            in_dev_put(in_dev);
111.        }
112.        return 0;
113. }
```

代码段 3-29 fib_check_nh 函数



图表 3-25 fn_hash_insert 之后 fib_table 和 fn_zone 及 fib_node 之间的关系

让我们放大 fib_node 的结构，并将其与 fib_zone 联系起来，于是得到下面这张图：



图表 3-26 fib_node 与 fib_alias、fib_info、fib_nh 结构的关系

通过这么一段时间的研究，可以这样理解这些结构之间的关系：

1,fib_table 中包含 fn_hash 结构指针

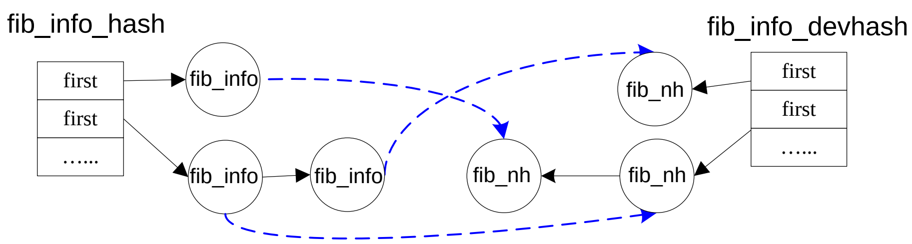
2,fn_hash 中包含 fn_zone 的数组，按照目的地址长度进行分类,相同长度的地址共用一个 fz_zone

3, fn_key 相同的两条路由（同一子网），共享一个路由节点（fn_node）

4,根据具体子网内地址的不同，使用不同的 fib_alias 和 fib_info

5,目的地址相同的情况下，也可以使用多条路由，不同的路由存放在不同的 fib_nh 里面

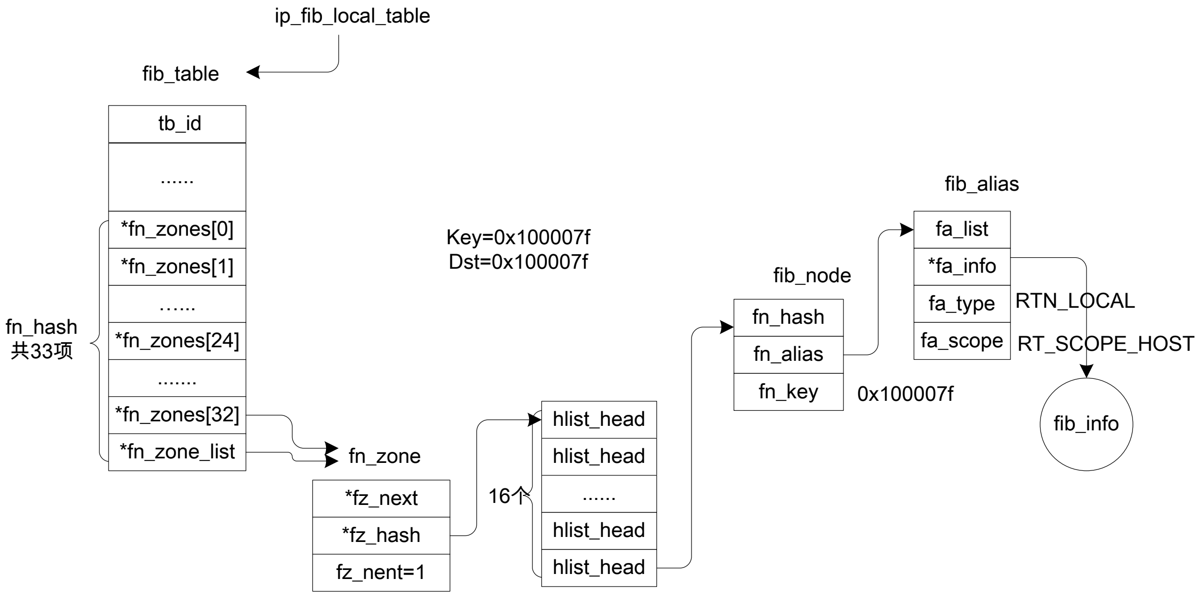
fib_create_info 函数中牵涉到 3 个 hash 表：fib_info_hash、fib_info_laddrhash、fib_info_devhash，为了简化讨论，我们不考虑第二个表，那么第二个和第三个表的关系如下：



图表 3-27 fib_info_hash 和 fib_info_devhash 的关系

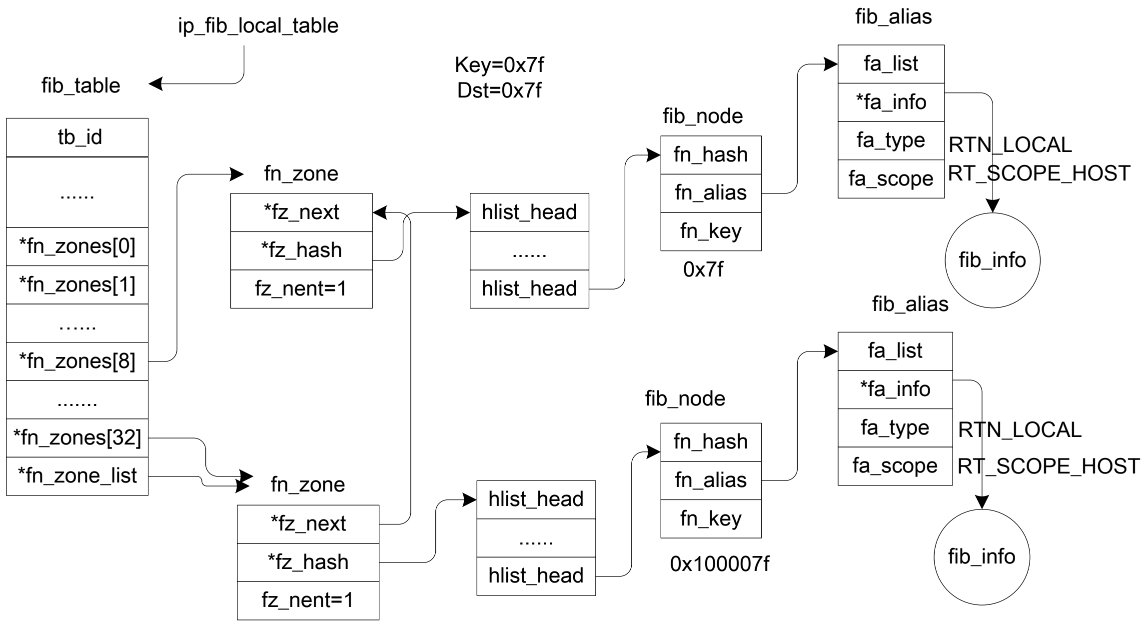
可以看出，fib_info 是指向 fib_nh 的，而它们分别被放到 2 个 hash 表中。

第①次fib_magic完成的结果：



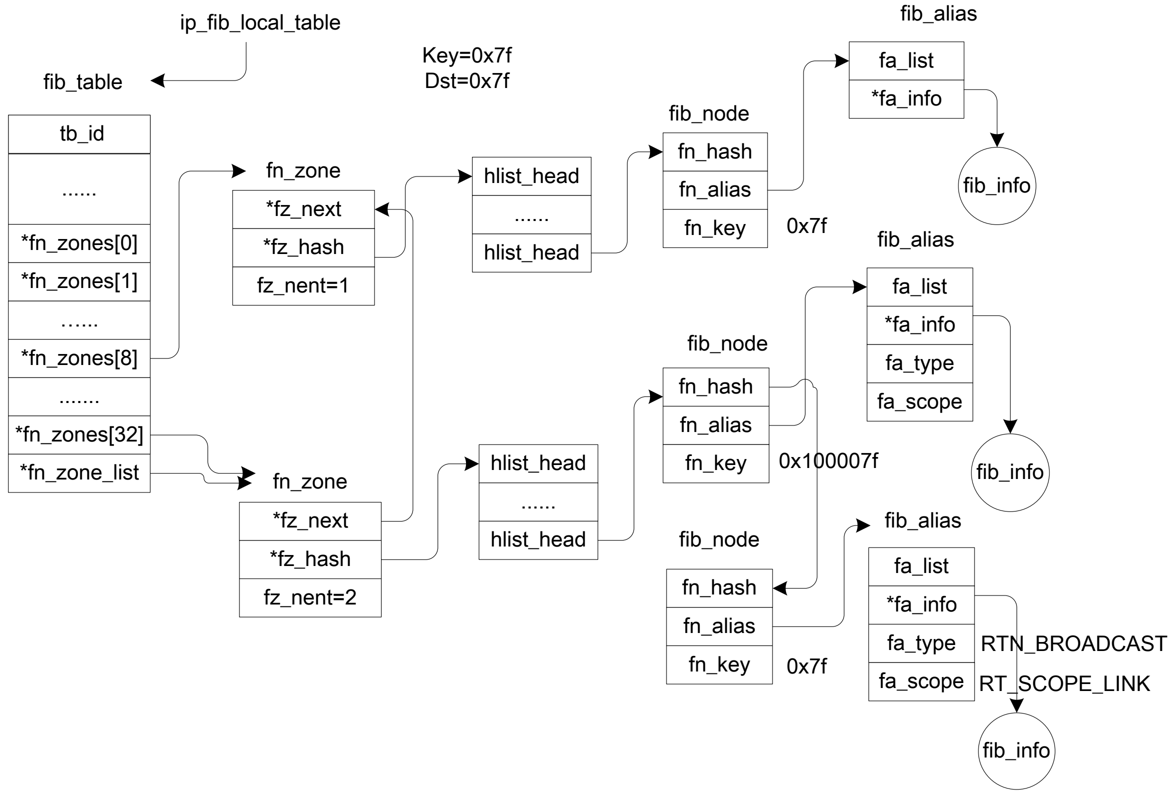
图表 3-28 第一次完成 FIB 表插入

第③次fib_magic完成的结果：



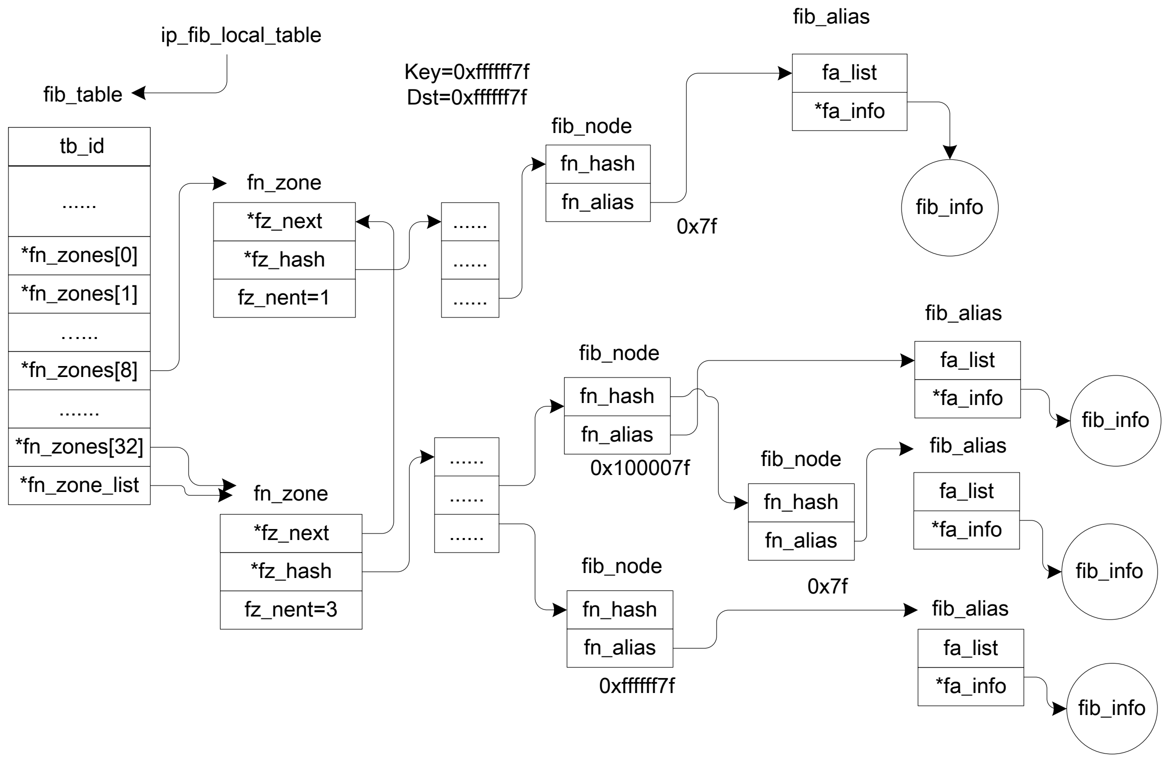
图表 3-29第二次完成 FIB 表插入

第④次fib_magic完成的结果：



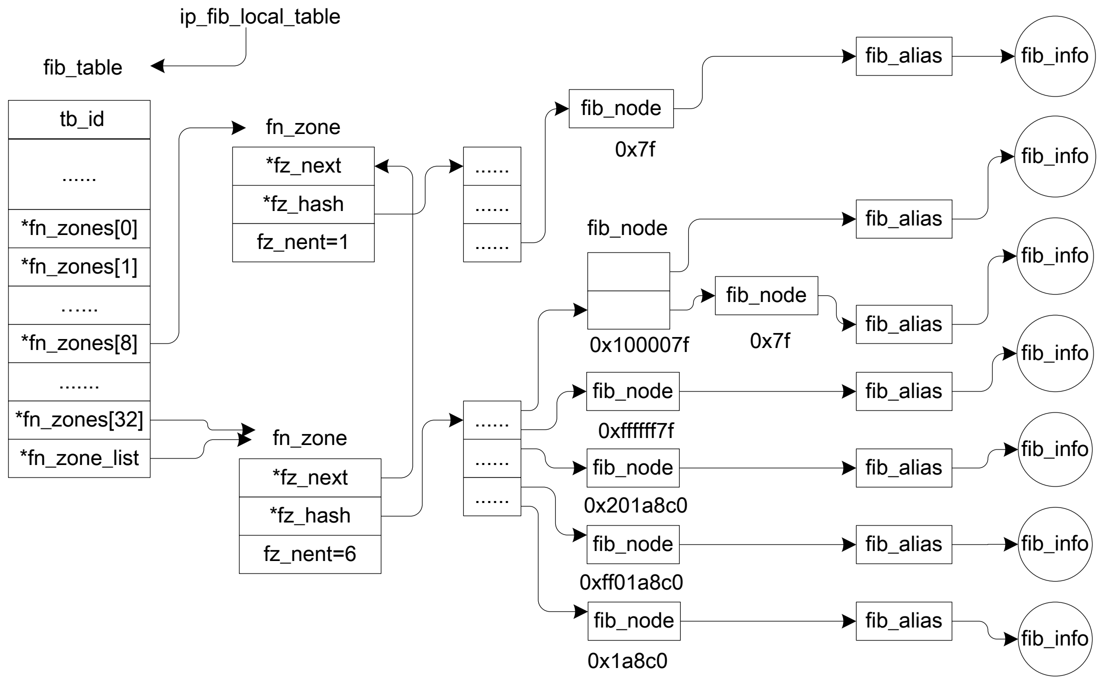
图表 3-30第三次完成 FIB 表插入

第⑤次fib_magic完成的结果：



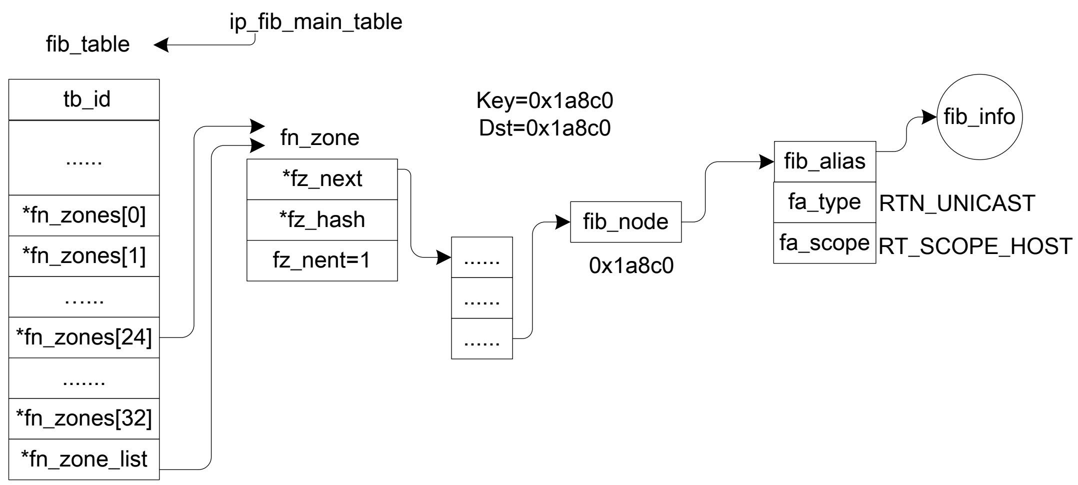
图表 3-31第四次完成 FIB 表插入

当配置例子中的接口 IP 地址时，会产生如下的结果：



图表 3-32 完成 FIB 表插入

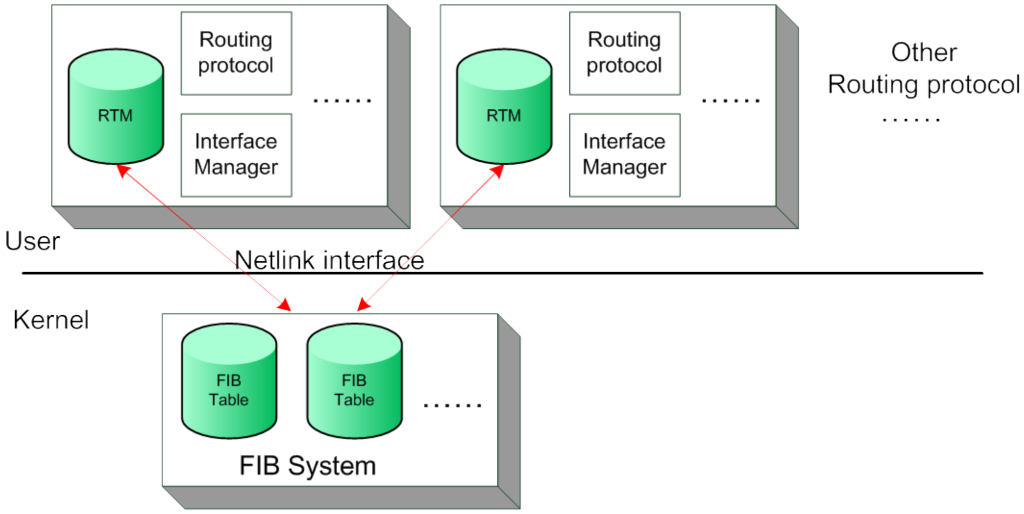
在上图发现什么问题没有？前文说过，loopback接口的IP地址设置有 4 次fib_magic，普通设备的IP地址设置有 5 次，不过最后一次不会再创建node了。所以在上图中应该会发现有 8 个node，但是只有 7 个。为什么会这样？这是因为普通设备的流程和loopback接口不同的是在第③次fib_magic里面访问的是main表而不是local表，所以这个node放到了main表：



图表 3-33 对 main FIB 表插入

3.5 直接访问路由表

前两节我们已经通过配置设备接口的 IP 地址间接的改写了路由数据库中的内容，那么还有其他方式访问路由数据库吗？答案是肯定的，目前所有基于 Linux 的路由系统基本属于下面的架构：



图表 3-34 常见路由软件架构

路由协议采用的 netlink 接口来更新 FIB 表，在本书中不可能再用一种路由协议去示例访问路由表的方式，我们可以采用类似的方式，比如静态配置路由表的方式来介绍 netlink 内部的实现。

在 Linux 上有两种访问路由表系统的命令

第一种： 使用 route 命令配置路由表

示例 1:添加到主机路由

route add –host 192.168.18.2 dev eth0

route add –host 192.168.183 gw 192.168.18.1

示例 2:添加到网络的路由

route add –net 10.10.10.10 netmask 255.255..0.0 eth0

route add –net 20.20.20.20 netmask 255.255..0.0gw 192.168.18.1

route add –net 30.30.30.30/24 eth1

示例 3:添加默认网关

route add default gw 192.168.18.1

示例 4:删除路由

route del –host 192.168.18.1 dev eth0

第二种： 使用 ip route 命令

示例 1: 设置到网络 10.0.0/24 的路由经过网关 192.168.18.1

ip route add 10.10.10.0/24 via 192.168.18.1

示例 2: 修改到网络 10.10.10.0/24 的直接路由，使其经过设备 eth0

ip route chg 10.10.10.0/24 dev eth0

使用 route 命令则会调用 ioctl，内核中会进入到 ip_rt_ioctl 函数。

```
1.  int ip_rt_ioctl(unsigned int cmd, void __user *arg)
2.  {
3.  int err;
4.  struct kern_rta rta;
5.  struct rtentry  r;
6.  struct {
7.      struct nlmsghdr nlh;
```

```
8.       struct rtmsg    rtm;
9.   } req;
10.
11. switch (cmd) {
12. case SIOCADDRT:     /* Add a route */
13. case SIOCDELRT:     /* Delete a route */
14.
15.     copy_from_user(&r, arg, sizeof(struct rtentry));
16.
17.     err = fib_convert_rtentry(cmd, &req.nlh, &req.rtm, &rta, &r);
18.     if (err == 0) {
19.         if (cmd == SIOCDELRT) {
20.             struct fib_table *tb = fib_get_table(req.rtm.rtm_table);
21.             err = tb->tb_delete(tb, &req.rtm, &rta, &req.nlh, NULL);
22.         } else {
23.             struct fib_table *tb = fib_new_table(req.rtm.rtm_table);
24.             err = tb->tb_insert(tb, &req.rtm, &rta, &req.nlh, NULL);
25.         }
26.         kfree(rta.rta_mx);
27.     }
28.     return err;
29. }
30. return -EINVAL;
31. }
```

代码段 3-30 ip_rt_ioctl 函数

这段代码和 fib_magic 函数很。

除了给接口分配 IP 地址会造成 FIB 表数据变化之外，还有另外一种方式，就是通过 rtnetlink 方式给内核增加路由。谁在使用这个接口？在一台路由器上任何路由协议都可以使用，比如 OSPF、RIP 等，当它们完成一条路由的计算之后，就会把这条路由加入到内核中，我们前面曾介绍过 rtmsg 的内容，而 rtnetlink 方式也是借助这种数据结构给内核下达增删路由的命令。不过不是通过 fib_magic，而是转到 inet_rtm_newroute 函数，其中再调用 tb->tb_insert( )来完成，其方式与 fib_magic 类似。只不过我们可以随意指定 fib_info 的部分字段了，比如 fib_info→ fib_protocol=rtmmsg→ protocol，可以是列表中的值。

```
1.  int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr* nlh, void *arg)
2.  {
3.      struct fib_table * tb;
4.      struct rtattr **rta = arg;
5.      struct rtmsg *r = NLMSG_DATA(nlh);
6.
7.      inet_check_attr(r, rta);
8.
9.      tb = fib_new_table(r->rtm_table);
10.     return tb->tb_insert(tb, r, (struct kern_rta*)rta, nlh, &NETLINK_CB(skb));
11. }
```

代码段 3-31 inet_rtm_newroute

这段代码是不是也和 fib_magic 函数很象啊？所以，对于路由协议如何更新 FIB 表的介绍就到这里，读者可以自行琢磨。

3.6 接口状态变化的处理过程

上一节介绍了配置系统的工作流程，给系统分配一个 IP：192.168.1.1，协议栈做了什么？首先它调用 ip_route_connect →ip_route_output_key→ ip_route_output_slow，然后创建一个邻居表项。

(用 ifconfig eth0 del 192.168.1.1 0 删除这个网卡的 IP 地址)

事情到这里好像都完美无缺，然后我们可以 ping 外部主机了。等等，好像有些东西没有提及。

因为在大多数情况，我们主机的以太网口都插上了网线，也就是说，如果按照上面的分析，似乎协议栈已经准备好了，但我们到现在还没有提到 neighbour 系统的设置，难道不需要这个子系统的协助吗？下面我们就来一步一步研究这个问题的答案。

我们已经知道在 Linux2.6 下每个网络设备驱动程序的设备实体(net_device)都有一个 priv 指针，指向了设备独有的数据块。虽然每种设备独有的数据结构有点区别，但是绝大多数都有一个 timer 结构，loopback 设备没有这个结构，为什么呢？分析设备驱动的初始化例程即在 dev→open 函数中发现这个 timer 的时间处理函数指针都指向了各自的所谓 watchdog 函数。熟悉嵌入式开发的读者可能要在此处迷惑，watchdog 一般都是值防止某个应用程序出现长时间占用 CPU 而提出的概念，外什么要这这里讨论 watchdog 呢？原因在于此 dog 非彼 dog。在网络设备驱动里，这个 watchdog 就只是用来轮询接口状态的定时器函数。Loopback 不需要检测链路状态，所以，它就没有这个函数，读者们明白了吗？

我们来看看这个定时器是如何工作的。当设备管理器调用某个网卡驱动 dev→open 例程的时候，就给这个驱动指定一个定时器，其处理函数约定熟成的叫做 xxx_watchdog，当然也有某些驱动程序开发者为了避免概念混淆，只给它起了一个很普通的名字，比如 xxx_timer。在实现这个处理函数中，一般有这样的惯例：

第5步． 轮询时间一般在 1 秒以上

第6步． 读取设备芯片关于网口的寄存器，检查其是否变化

第7步． 如果有变化，统一调用 netif_carrier_on 或 netif_carrier_off 来通知系统内部其他模块，比如邻居子系统。

由于每个驱动程序的实现不一样，而定时器这一部分相对一致，我们就不仔细研究设备驱动程序的实现了。现在假设我们有一台没有接网线的主机，当我们插上另一端已经有主机的网线，当 watchdog 定时器扫描发现已经链路已经有信号时，就会调用 netif_carrier_on 函数，其实现如下：

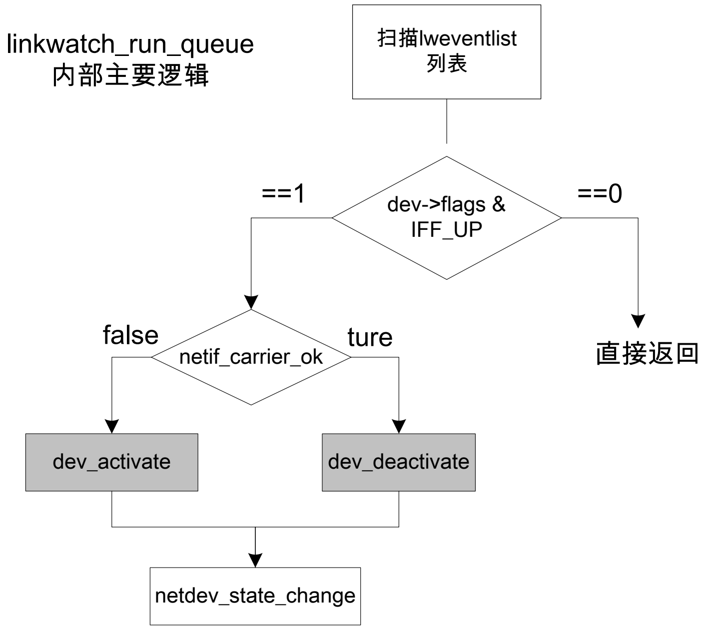
```
1.  void netif_carrier_on(struct net_device *dev)
2.  {
    去掉 NO_CARRIER 标志，然后进入下面的函数。
3.  if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state))
4.      linkwatch_fire_event(dev);
     如果设备没有被卸载，那么还得继续定时器扫描
5.  if (netif_running(dev))
6.      __netdev_watchdog_up(dev);
7.  }
```

代码段 3-32 netif_carrier_on 函数

linkwatch_fire_event 函数比较复杂，我们在这里不打算列出其代码，不过还是给读者它完成的工作：

1. 创建一个 lw_event{}结构，表示一个事件。
2. 把这个结构挂到 lweventlist 工作队列上。
3. 调用 schedule_work 或 schedule_delayed_work（如果当事件太频繁）促使内核对 lweventlist 扫描并执行该节点上回调函数——linkwatch_event。

当系统有时间处理工作队列时，执行 linkwatch_event：



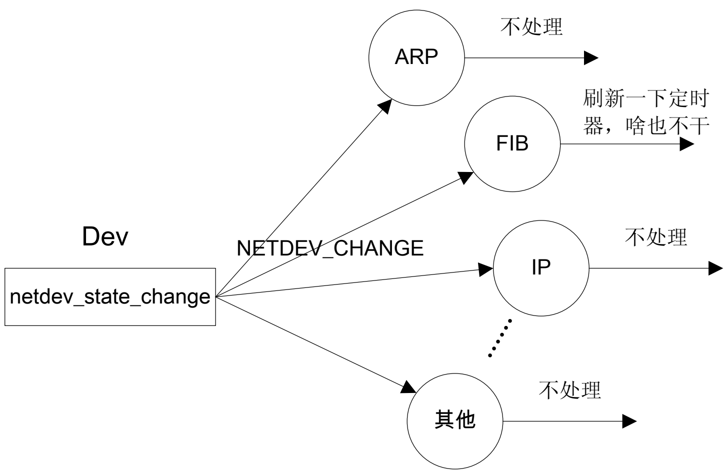
图表 3-35linkwatch_run_queue 内部主要逻辑

之所以要将图中两个函数设为灰底，是因为在将来的设备发送/接收过程中要接触到这两个函数的工作。现在我们先放在一边，看看 netdev_state_change。此函数比较简单，但是非常重用，它就是设备连接邻居子系统的通道。它就是调用了下面这行代码：

```
raw_notifier_call_chain (&netdev_chain, NETDEV_CHANGE, dev);
```

到此为止，驱动程序的任务基本完成，接下来要看看这行代码到底做了什么事。

驱动程序最终发送了一个 notify，其接收对象 netdev_chain 链表，凡是对网络设备事件感兴趣的模块，都要挂在这个链表上。搜遍整个源代码，可以看到挂接到此链表的子系统挺多的，但是，对 NETDEV_CHANGE 这个事件感兴趣的，嗯，几乎没有。



图表 3-36 NETDEV_CHANGE 事件

从上图可以看出，没有人对设备的状态的变化感兴趣，所以，可以回答前面的问题了：网卡接口是否接通与 ARP 工作与否或其他子系统没有直接关系。也就是说，如果网口从物理 down 变成物理 up，不会触发 ARP 或路由系统的变化。

第4章　网络层实现的初步研究

4.1　从 Ping 127.0.0.1 开始旅程

Ping 是潜水艇人员的专用术语，表示回应的声纳脉冲，在网络中 Ping 是一个十分好用的 TCP/IP 工具。它主要的功能是用来检测网络的连通情况和分析网络速度。

我们先给出一段 ping 的代码基本实现，出于篇幅的原因，去掉了一些不重要的语句，让我们只关心那些关键的代码吧：

```
1.  #define PACKET_SIZE    4096
2.  #define MAX_WAIT_TIME   5
3.
4.  char sendpacket[PACKET_SIZE];
5.  char recvpacket[PACKET_SIZE];
6.  int sockfd,datalen=56;
7.  int nsend=0,nreceived=0;
8.  struct sockaddr_in dest_addr;
9.  pid_t pid;
10. struct sockaddr_in from;
11. struct timeval tvrecv;
12.
13. void statistics()
14. {
15.       printf(...... );/*打印统计信息*/
16.       close(sockfd);
17.       exit(1);
18. }
19. /*校验和算法*/
20. unsigned short cal_chksum(unsigned short *addr,int len)
21. {      ......
22. }
23. /*设置 ICMP 报头*/
24. int pack(int pack_no)
25. {
26.       int i,packsize;
27.       struct icmp *icmp;
28.       struct timeval *tval;
29.
30.       icmp=(struct icmp*)sendpacket;
31.       icmp->icmp_type=ICMP_ECHO;
32.       icmp->icmp_code=0;
33.       icmp->icmp_cksum=0;
34.       icmp->icmp_seq=pack_no;
35.       icmp->icmp_id=pid;
36.       packsize=8+datalen;
37.       tval= (struct timeval *)icmp->icmp_data;
38.       gettimeofday(tval,NULL);    /*记录发送时间*/
39.       /*校验算法*/
40.       icmp->icmp_cksum = cal_chksum((unsigned short *)icmp, packsize);
41.       return packsize;
42. }
43.
44. /*发送三个 ICMP 报文*/
45. void send_packet()
46. {
47.       int packetsize;
48.       while( nsend < 3)
49.       {
50.              nsend++;
```

```
51.             packetsize = pack(nsend); /*设置 ICMP 报头*/
52.             sendto(sockfd,  sendpacket,  packetsize,  0,
53.                      (struct sockaddr *)&dest_addr,sizeof(dest_addr))
54.
55.             sleep(1); /*每隔一秒发送一个 ICMP 报文*/
56.       }
57. }
58.
59. /*接收所有 ICMP 报文*/
60. void recv_packet()
61. {
62.       int n,fromlen;
63.
64.       while( nreceived < nsend)
65.       {
66.             alarm(MAX_WAIT_TIME);
67.             recvfrom(sockfd,recvpacket,sizeof(recvpacket),0,
68.                       (struct sockaddr *)&from,&fromlen); /*显示相关信息*/
69.
70.             gettimeofday(&tvrecv,NULL);  /*记录接收时间*/
71.             unpack(recvpacket, n) /*对该 ICMP 报文进行解析*/
72.             nreceived++;
73.       }
74.
75. }
76. /*剥去 ICMP 报头*/
77. int unpack(char *buf,int len)
78. {      int i,iphdrlen;
79.       struct ip *ip;
80.       struct icmp *icmp;
81.       struct timeval *tvsend;
82.       double rtt;
83.
84.       ip = (struct ip *)buf;
85.       iphdrlen = ip->ip_hl<<2;    /*求 ip 报头长度,即 ip 报头的长度标志乘 4*/
86.       icmp = (struct icmp *)(buf+iphdrlen);  /*越过 ip 报头,指向 ICMP 报头*/
87.       len -= iphdrlen;           /*ICMP 报头及 ICMP 数据报的总长度*/
88.       if ( len < 8)              /*小于 ICMP 报头长度则不合理*/
89.       {      ......
90.             return -1;
91.       }
92.       /*确保所接收的是我所发的的 ICMP 的回应*/
93.       if( (icmp->icmp_type == ICMP_ECHOREPLY) && (icmp->icmp_id == pid) )
94.      {
95.             ......
96.             printf（...... ); /*显示相关信息*/
97.       }
98.       else
99.          return -1;
100.    }
101.
102.    main(int argc,char *argv[])
103.    {
104.          unsigned long inaddr=0l;
105.          int size=50 * 1024;
106.
107.          /*生成使用 ICMP 的原始套接字, 其 protocol 是 17*/
108.          sockfd=socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
109.
110.          /*扩大套接字接收缓冲区到 50K 这样做主要为了减小接收缓冲区溢出的
111.            的可能性,若无意中 ping 一个广播地址或多播地址,将会引来大量应答*/
112.          setsockopt(sockfd,SOL_SOCKET,SO_RCVBUF,&size,sizeof(size) );
113.          bzero(&dest_addr,sizeof(dest_addr));
114.          dest_addr.sin_family=AF_INET;
115.
116.          send_packet();  /*发送所有 ICMP 报文*/
```

```
117.          recv_packet();  /*接收所有 ICMP 报文*/
118.          statistics(); /*进行统计*/
119.
120.          return 0;
121.}
```

代码段 4-1 ping 的伪代码

从上面的代码可以看出，ping 使用了 SOCK_RAW 的方式调用 socket 系统接口。SOCK_RAW 的含义即基于 IP 层协议建立的通信机制。

Ping 回送地址是为了检查本地的 TCP/IP 协议有没有设置好；Ping 本机 IP 地址，这样是为了检查本机的 IP 地址是否设置有误；Ping 本网网关或本网 IP 地址，这样的是为了检查硬件设备是否有问题，也可以检查本机与本地网络连接是否正常（在非局域网中这一步骤可以忽略）；Ping 远程 IP 地址，这主要是检查本网或本机与外部的连接是否正常。

如果连本地址无法 Ping 通，则表明本地机 TCP/IP 协议不能正常工作。

4.2 再次相遇 Socket 系统调用

回想一下 socket 调用到 inet_create 的时候，当如果是 RAW 应用的话，就会有如下的代码片段，

```
1.  static int inet_create(struct socket *sock, int protocol)if (SOCK_RAW == sock->type)
    {
        ......
    这几句赋值很有意义，我们要在 send 的实现代码中经常看到 inet→hdrincl 这个变量，记住，在目前的
    ping 应用中，这个值是 0，因为我们应用层设置的 protocol 是 IPPROTO_ICMP。还有，num 也被赋予
    一个值，大家应该记住，num 其实就是 TCP/UDP 中提到的端口号，不过在 RAW IP 中这个 num 表示一种
    协议而已，不是真正的“端口号”。
2.      inet->num = protocol;
3.      if (IPPROTO_RAW == protocol)
4.          inet->hdrincl = 1;
         ......
    这里 sk_protocol 是 IPPROTO_ICMP
5.  sk->sk_protocol    = protocol;
    下面执行的是 raw_prot 中的 raw_init 函数
6.  if (sk->sk_prot->init) {
7.      err = sk->sk_prot->init(sk);
         ......
8.  }
```

代码段 4-2 raw socket 情况下的 inet_create 函数

再次碰到的 socket 函数参数发生了变化，和前一章机会没有变化，只是在 inet_create 函数中创建 socket 时 sock→ops 指向了 inet_sockraw_ops，而 sk→sk_prot 指向了 raw_prot。相应的 sk→sk_prot→init 被执行，不过它不能满足读者的求知欲，因为它内部没有什么可以研究的东西。

4.3 IP 数据报文格式

与其他书籍将报文帧结构放在附录的安排不一样，我把协议报文的数据结构放在前面，这说明什么？说明：帧结构是非常重要的一部分，对于网络协议，报文帧结构就是核心，所有的代码都围绕这些结构来运行。每个读者，都必须先把报文帧结构熟记于心，当然，如果实在记不住，翻到这一页就可以了。

IP 协议用于管理客户端和服务器端之间的报文传送。普通的 I P 首部长为 2 0 个字节，除非含

有选项字段。

I P 头结构如下：

<table>
<tr><td>4b 版本</td><td>4b 首部长度</td><td>4b TOS</td><td colspan="2">16b 总长度（字节数）</td></tr>
<tr><td colspan="3">16b 标识</td><td>3b 标志</td><td>13b 段偏移</td></tr>
<tr><td colspan="2">8b TTL</td><td>8b 协议</td><td colspan="2">16b 头部校验和</td></tr>
<tr><td colspan="5">32b 源IP</td></tr>
<tr><td colspan="5">32b 目的IP</td></tr>
<tr><td colspan="5">选项（如果有）</td></tr>
<tr><td colspan="5">Payload</td></tr>
</table>

图表 4-1 IP 层数据报文格式

分析图 4- 1 中的首部。最高位在左边，记为 0 bit；最低位在右边，记为 31 bit。

4 个字节的 32 bit 值以下面的次序传输：首先是 0～7 bit，其次 8～15 bit，然后 1 6～23 bit，

最后是 24～31 bit。这种传输次序称作 big endian 字节序。由于 T C P / I P 首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。以其他形式存储二进制整数的机器，如 little endian 格式，则必须在传输数据之前把首部转换成网络字节序。目前的协议版本号是 4，因此 I P 有时也称作 IPv4。

首部长度指的是首部占 32 bit 字的数目，包括任何选项。由于它是一个 4 比特字段，因此首部最长为 6 0 个字节。普通 I P 数据报（没有任何选择项）字段的值是 5。

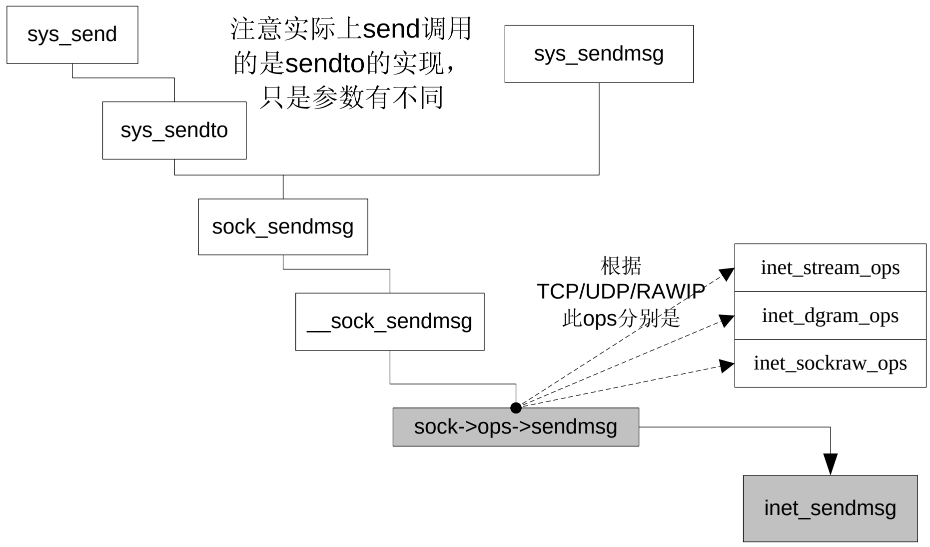
服务类型（TOS）字段包括一个 3 bit 的优先权子字段（现在已被忽略），4 bit 的 TO S 子字段和 1 bit 未用位但必须置 0。4 bit 的 TO S 分别代表：最小时延、最大吞吐量、最高可靠性和最小费用。4 bit 中只能置其中 1 bit。如果所有 4 bit 均为 0，那么就意味着是一般服务。RFC 1340[Reynolds and Postel 1992] 描述了所有的标准应用如何设置这些服务类型。RFC 1349[Almquist 1992]对该 RFC 进行了修正，更为详细地描述了 TOS 的特性

4.4 send 系统调用

这个系统调用我们之前研究配置的时候没有碰到，现在总算与之相遇，现在我们就开始研究发送的具体流程吧。

我们知道 socket 应用程序一般使用 send 或 sendto 系统调用发送数据，那么它们到底什么区别呢？下面的图会给你一些答案。

（注意，过去的内核版本是单独把 send 作为一个系统调用放在系统调用表中，现在是统一经过 sys_socketcall 系统调用接口转进来。不过对于我们没有什么区别，同样的，之后要介绍的 recv、bind、listen 等函数都如此）



图表 4-2 sys_send 函数调用树

在用户态调用的系统接口 send 和 sendto 及 sendmsg，其实都调用了 sock_sendmsg，而且用得较多的 send 调用的 sendto，只是参数要注意，内核代码如下：sys_sendto(fd, buff, len, flags, NULL, 0);即最后两个参数为 0（NULL）。

在 BSD Socket 层使用 msghdr{}结构保存数据，在 INET Socket 层以下都使用 sk_buff{ }结构保存数据。msghdr{ }结构是出于对 4.4BSD 的兼容性而定义的，内核内部函数 sock_sendmsg（），sock_recvmsg（）都使用这个结构。其定义如下：

```
1.  /*
2.   *  As we do 4.4BSD message passing we use a 4.4BSD message passing
3.   *  system, not 4.3. Thus msg_accrights(len) are now missing. They
4.   *  belong in an obscure libc emulation or the bin.
5.   */
6.  
7.  struct msghdr
8.  {
9.      void *   msg_name;   /* Socket 名称，一般用 NULL 初始化     */
10.     int      msg_namelen;   /* 上面名称的长度     */
11.     struct iovec *  msg_iov;    /* 发送或接收的数据块结构          */
12.     __kernel_size_t msg_iovlen; /* 数据块的个数      */
13.     void    *   msg_control; /* 每个协议一个 magic 数 (eg BSD 传递文件描述符) */
14.     __kernel_size_t msg_controllen; /* cmsg 链表的长度*/
15.     unsigned msg_flags;
16. };
```

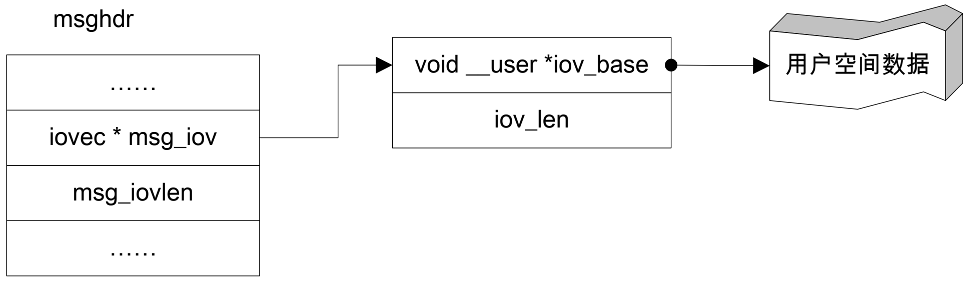
代码段 4-3 msghdr 结构

msg_name 和 msg_namelen 就是数据报文要发向的对端的地址信息（即 sendto 系统调用中的 addr 和 addr_len)，当使用 send 时，它们的值为 NULL 和 0。

```
1.  /* A word of warning: Our uio structure will clash with the C library one (which is
    now obsolete). Remove the C library one from sys/uio.h if you have a very old library
    set */
2.  
3.  struct iovec
4.  {
5.      void __user *iov_base;  /* BSD uses caddr_t (1003.1g requires void *) */
6.      __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
7.  };
```

代码段 4-4 iovec 结构

此结构表示存放待发送数据的一个缓冲区，iov_base 是缓冲区的起始地址，指向用户层待发送数据, iov_len 是缓冲区的长度。msg_iovlen 是缓冲区的数量，对于 sendto 和 send 来讲，msg_iovlen 都是 1。msg_flags 即为传入的参数 flags，现在暂时不过多的关注 flags 的应用。msg_control 和 msg_controllen 暂时不关注。



图表 4-3 meghdr 如何指向用户空间数据

参照图 4-2，看到虽然 TCP/UDP/RAWIP 的 ops 分别有一个，这里的 RAW IP 对应的 ops 是 inet_sockraw_ops，其对应的 sendmsg 函数指针成员都指向 inet_sendmsg，而且其它两种协议的 sendmsg 函数指针也指向到这个函数。

```
1.  int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg, size_t
    size)
2.  {
3.      struct sock *sk = sock->sk;
4.  
5.  /* 我们可能要绑定这个 socket. 这里要提醒读者的是 RAW IP 也有自己的 bind 函数，只不过此 bind 非
    彼 bind，由于在 inet_create 函数中针对 RAW socket 曾经做了 inet->num = protocol 这么一个动
    作，而在 ping 应用程序中 protocol 是 IPPROTO_ICMP（1），所以 num＝1，于是下面这句话不会被执行。
    */
6.      if (!inet_sk(sk)->num && inet_autobind(sk))
7.          return -EAGAIN;
8.  
9.      return sk->sk_prot->sendmsg(iocb, sk, msg, size);
10. }
```

代码段 4-5inet_sendmsg 函数

什么情况下会执行 inet_autobind 函数呢？就是当创建 UDP/TCP 的 socket 后即调用 send 函数发送数据（这是一般客户端代码的行为方式），于是就会调用到此函数，因为之前的 inet_create 函数没有给这种类型的 socket 指定 num。所以凡是 TCP/UDP 的客户端必定会进入到此函数中。那么 inet_autobind 到底干嘛呢？它里面调用了 sk->sk_prot->get_port，很明显，由于 RAW IP 没有实现 get_port 函数，所以只能是调用 udp/tcp_v4_get_port 函数获得未使用的 port 号赋给 inet->num，然后又把它赋给 inet->sport 成员。我们会在 UDP 一章中继续了解。

上文已经说过，由于是 raw 类型的 socket，所以实际指向 raw_prot 结构。那么相应的 sendmsg 函数指针就是 raw_sendmsg 函数。

对于使用 RAW 选项打开的 socket，其对应的函数是 raw_prot→raw_sendmsg，

```
1.  static int  raw_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
    size_t len)
2.  {
3.      struct inet_sock *inet = inet_sk(sk);
```

```
4.      struct ipcm_cookie ipc;
5.      struct rtable *rt = NULL;
6.      int free = 0;
7.      u32 daddr;
8.      u32 saddr;
9.      u8  tos;
10.     int err;
11.
12.     ......
13.
14.     /*
15.      *  Get and verify the address.
16.      */
17.
18.     if (msg->msg_namelen) {
19.         struct sockaddr_in *usin = (struct sockaddr_in*)msg->msg_name;
20.         ......
21.         daddr = usin->sin_addr.s_addr;
22.         ......
23.     } else {
24.         err = -EDESTADDRREQ;
25.         if (sk->sk_state != TCP_ESTABLISHED)
26.             goto out;
27.         daddr = inet->daddr;
28.     }
29.
30.     ipc.addr = inet->saddr;
31.     ipc.opt = NULL;
32.     ipc.oif = sk->sk_bound_dev_if;
33.
34.     if (msg->msg_controllen) {
35.         err = ip_cmsg_send(msg, &ipc);
36.         ......
37.     }
38.
39.     saddr = ipc.addr;
40.     ipc.addr = daddr;
41.
42.     if (!ipc.opt)
43.         ipc.opt = inet->opt;
44.
45.     if (ipc.opt) {
46.         err = -EINVAL;
47.         /* Linux 不会处理 raw socket 的头，所以 IP 选项+ IP_HDRINCL 没有意义
48.          */
49.         if (inet->hdrincl)
50.             goto done;
51.         if (ipc.opt->srr) {
52.             if (!daddr)
53.                 goto done;
54.             daddr = ipc.opt->faddr;
55.         }
56.     }
57.     tos = RT_CONN_FLAGS(sk);
       msg->msg_flags 在此 ping 应用中是 0，如果曾经设置了 MSG_DONTROUTE，会对后面的路由查找有
       影响
58.     if (msg->msg_flags & MSG_DONTROUTE)
59.         tos |= RTO_ONLINK;
60.
61.     ......
62.
63.     {
64.         struct flowi fl = { .oif = ipc.oif,
65.                     .nl_u = { .ip4_u =
66.                           { .daddr = daddr,  /*目的地址*/
67.                          .saddr = saddr,     /*源地址*/
68.                          .tos = tos } },
          如果包含 IP 头,就是普通的 RAW 类型,否则就是用户定义的类型,在创建 socket 的时候(raw socket
```

```
        情况下 inet_create 函数的第 3、4 行)我们已经设置 inet->hdrincl 为 0，所以.proto 就是
        sk_protocol，它将被写到报文头中。
        在此 sk_protocol 是 IPPROTO_ICMP 报文，其值是 17
69.                    .proto = inet->hdrincl ? IPPROTO_RAW :
70.                                 sk->sk_protocol,
71.                  };
      此函数只对 IPPROTO_ICMP 感兴趣，正好合乎我们目前的应用，它将用户空间的 icmp type 类型拷贝
      到内核空间的 fl 结构中
72.         if (!inet->hdrincl)
73.             raw_probe_proto_opt(&fl, msg);
      对 flowi 结构填入了必要的信息，然后传入下面的函数，并将路由模块计算出的信息放入 rt 结构中。
      注意，rt 是在该函数内部（具体在__mkroute_output，下面会讲）创建，所以必须传&rt 而不是 rt。
74.         err=ip_route_output_flow(&rt,&fl,sk,!(msg->msg_flags&MSG_DONTWAIT));
75.     }
76.     ......
77.
78.     err = -EACCES;
79.     if (rt->rt_flags & RTCF_BROADCAST && !sock_flag(sk, SOCK_BROADCAST))
80.         goto done;
81.
82.     if (msg->msg_flags & MSG_CONFIRM)
83.         goto do_confirm;
84. back_from_confirm:
    对于普通的 IP 层应用来说，应该是执行下面这行代码，但是对于 ping 应用而言，执行的却是 98 行的代
    码
85.    if (inet->hdrincl)
86.         err = raw_send_hdrinc(sk, msg->msg_iov, len,
87.                     rt, msg->msg_flags);
88.
89.    else {
    我们的 ping 代码要走到这里
90.     if (!ipc.addr)
91.         ipc.addr = rt->rt_dst;
92.     lock_sock(sk);
    将多个小的报文打包成一个大包，不过目前的情况还不需要，此函数还完成一个很重要的事情：申请一个
    skb 然后把它挂接到 sk_write_queue 队列中，然后由 105 行的函数发送出去。
93.     err = ip_append_data(sk, ip_generic_getfrag, msg->msg_iov, len, 0,
94.                 &ipc, rt, msg->msg_flags);
95.     if (err)
96.         ip_flush_pending_frames(sk);
97.     else if (!(msg->msg_flags & MSG_MORE))
     开始往 IP 层发送报文
98.         err = ip_push_pending_frames(sk);
     当 socket 被锁住，收到的包暂时不能放入接收队列中，他们只能放进 backlog 队列中等待进一步的处
     理。
99.     release_sock(sk);
100.        }
101.    done:
102.        if (free)
103.            kfree(ipc.opt);
104.
105.    out:
106.       ......
107.        return len;
108.
109.    do_confirm:
110.        dst_confirm(&rt->u.dst);
111.        if (!(msg->msg_flags & MSG_PROBE) || len)
112.            goto back_from_confirm;
113.        err = 0;
114.        goto done;
115.    }
```

代码段 4-6 raw_sendmsg 函数

ip_append_data 函数将许多小片的数据整合成一个足够大的报文发送出去，这可以稍微提高一

些性能，但是如果是使用 RAW IP 的特殊协议，使 inet→hdrincl 等于 1，那么这表示应用程序不希望内核对报文做过多干涉而要求直接发送出去，于是使用 raw_send_hdrinc 函数来完成。

4.5 在路由系统中游历

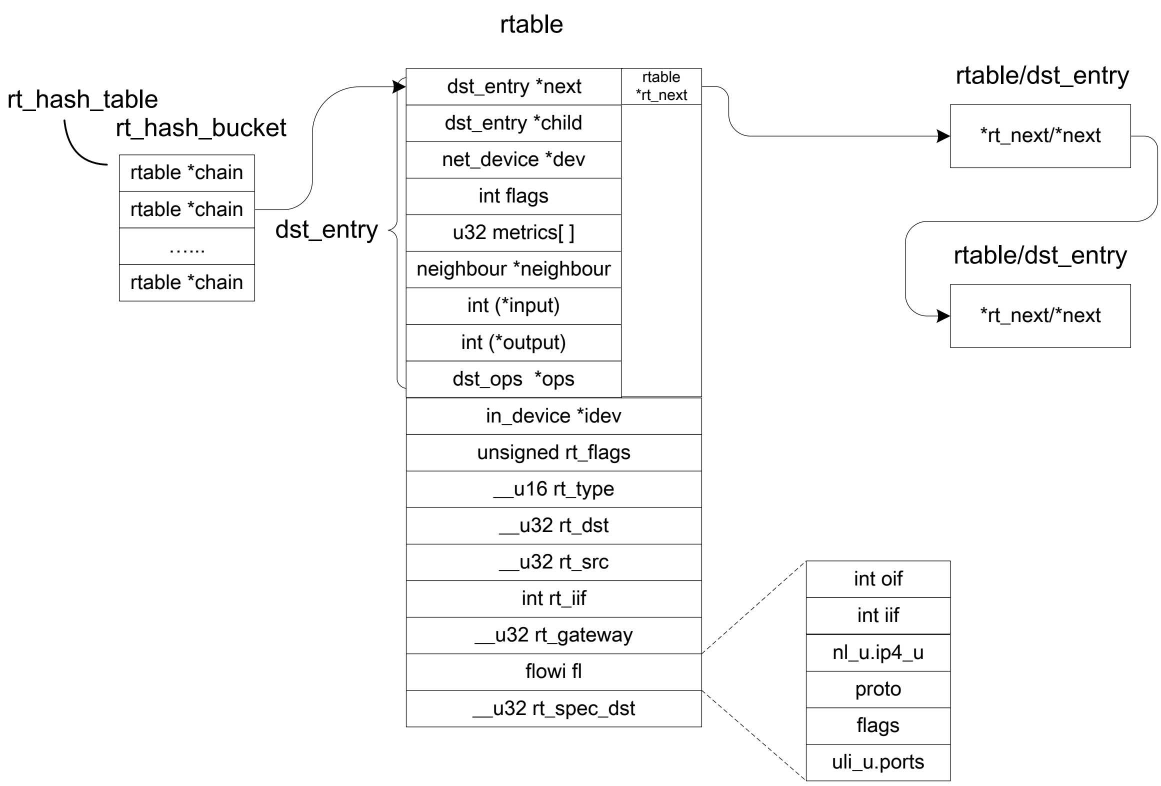
4.5.1 查找出口

当要发送一个报文时，必定要查询发送接口，这个过程被 Linux 分为 3 个步骤，第一个步骤是查询路由 cache，第二个步骤是查询 FIB 表，第三步是将查询结果填入路由 cache 中以便将来查询。现在来介绍一下路由 cache。

- **路由 cache**

当确定了一条路由时，路由表项就被放入路由 cache 中，这意味着一旦知道路由并放入 cache 后，经过同样路由的报文能够立即找到出口。一个报文在本地机器上可以有一个目的地址，它最终的目的也许是本地可达的主机，也可能被发送到下一跳节点。因此，路由和目的 cache 被设计成报文目的地址对实际的 IP 发送过程是透明的，目的 cache 表项可以和路由 cache 表项互换。为了指向指向目的 cache，同样的 dst 字段也指向路由表的表项。这让 IP 用有效的方法检查待发送报文的目的地址，而不用查找路由或显式的检查目的地址是否已经被解析到硬件地址。

路由 cache 可以被看作 FIB 的子集，它是用来优化已知目的地址的已打开 socket 的快速路由的。路由 cache 的实现基于通用的目的地址 cache 架构，它由 hash 表组成，每一个表项包含路由表项。这个表可以用简单 key 完成快速搜索。hash 表的实现允许冲突，因为每一个 hash 表位置可以包含多个路由。IP 中的路由 cache 是一个 hash 桶（即 rt_hash_bucket）的实例，叫 rt_hase_table，在此数组中每一个单元包含一个指向路由表的链，每个匹配某目的地址的路由放在一个由链表指向的连接列表中。其基本结构如下：



图表 4-4 rt_hash_table 和 rtable、dst_entry 的关系

路由表是路由 cache 中存放每个路由的基本数据结构。本质上它是面向对象的。这是把路由 cache 看作是来源于通用的目的地址 cache 功能的原因。例如，sk_buff 结构为外发报文包含一个指向目的 cache 表项的指针，这个 dst 表项为报文包含一个指向路由 cache 表项的指针。因此用相似的方法，这个 rtable 结构定义的首要的几个字段指向目的 cache，dst_entry。

```
1.  struct rtable
2.  {
        我们使用一个联合，使这个指向下一个 rtable 实例的指针即能通过一个指向目的 cache 表项的指针（dst）或指向路由表项指针（rt_next）访问。我们检查 skb 里的 dst 字段，它指向目的 cache 表项。如果不为 NULL，则意味着该 IP 包知道将包发往何处，如果为 NULL，则必须找到一个路由要么是内部路由，要么外部路由。
3.      union
4.      {
5.          struct dst_entry   dst;
6.          struct rtable       *rt_table;
7.      }u;
     Rt_flags 能被提供应用程序接口的路由表使用。因为在单个 hash 桶内也许有多个路由，那么这些路由会冲突。当垃圾回收程序处理这些路由 cache，如果和高价值的路由发生冲突时，低价值的路由倾向于被清除出去，路由控制 flags 决定这些路由的值。
8.      unsigned    rt_flags;
     Rt_type 是这个路由的类型，它确定这些路由是否单播、组播或本地路由。
9.      unsigned    rt_type;
     Rt_dst 是 IP 目的地址，rt_src 是 IP 源地址。Rt_iif 是路由输入接口索引。
10.     __u32    rt_dst;
11.     __u32    rt_src;
12.     Int     rt_iif;
13.     __u32   rt_gateway; //网关或邻居的 IP 地址。
14.     struct flowi    fl; //包含实际的 hash 键，
15.     __u32   rt_spec_dst;//为 UDP socket 用户设置源地址的特殊目的地址[RFC1122]
    Internet 对端结构被用来生成在 IP 头中的 16 位标识，它被叫做 long-living peer information，Linux 通过增加对每个指定的主机的每个发出去的报文的数量计算该 ID，对此路由由该 Internet 对端结构通过对端指针访问。
16.     struct inet_peer    *peer;
17. }
```

代码段 4-7 rtabl 数据结构

路由 cache 的搜索算法：先用一个简单的 hash code 定位 hash 桶里的一个 slot，然后用一个 key 去匹配一个指定的路由，这必须遍历这个 slot 所有的路由列表直到 rt_next 等于 NULL。第二级查找是把 rtable 表项中的 fl 字段和收到的报文中的信息进行精确匹配。fl 结构包含了能确定某路由的所有信息。

- **flowi 数据结构**

这里我们碰到一个奇怪的结构就是 flowi。从它的字面上来理解似乎是和“流”有关的一个东西，但源代码中没有对它进行详细解释。而且，BSD 协议栈中也没有这么一个结构。那么它到底是什么呢？其实，它就是标识一个发送/接收流的结构，不同用户的业务流之间是如何被内核区别就依靠它。所以，flowi 中的 i 可以理解成 identifier。首先，它的结构如下：

```
1.  struct flowi {
    下面两个字段确定 input 和 output 接口，lif 是输入接口索引，它是从 net_device 结构里的 ifIndex 获取的，这个 net_device 是指收到报文的设备。oif 是输出接口索引。通常，对于一个指定路由由 iif 或者 oif 会被赋值，而其他字段是 0。
2.      int  oif;
3.      int  iif;
    下面这个结构是通用的，所以我们用联合来定义 Ipv4，Ipv6 和 DECnet：
4.      union {
```

```
5.          struct {
6.              __u32           daddr;
7.              __u32           saddr;
8.              __u32           fwmark; 防火墙 mark，用来做流量 shaping。。。
    Tos 是 IP 头的 ToS 字段。IP 并没有定义 TOS 的 0 位，但 Linux 用它来定义一个直连的主机。这个 bit 也
    被 ARP 协议使用。
9.              __u8        tos; 定义了范围或该路由覆盖的概念性距离
10.             __u8        scope;      } ip4_u;
11.
12.         struct {
13.             ...... //Ipv6 的数据,不用理会
14.         } ip6_u;
15.
16.         struct {
17.          ...... //dnnet 的数据,不用理会
18.         } dn_u;
19.     } nl_u;
20.     ...... // Ipv6 和 dnnet 的数据,不用理会
21. #define fl4_dst     nl_u.ip4_u.daddr
22. #define fl4_src     nl_u.ip4_u.saddr
23. #define fl4_fwmark  nl_u.ip4_u.fwmark
24. #define fl4_tos     nl_u.ip4_u.tos
25. #define fl4_scope   nl_u.ip4_u.scope
26.
27.     __u8 proto;
28.     __u8 flags;
29. #define FLOWI_FLAG_MULTIPATHOLDROUTE 0x01
30.     union {
31.         struct {
32.             __u16   sport;
33.             __u16   dport;
34.         } ports;
35.
36.         struct {
37.             __u8 type;
38.             __u8 code;
39.         } icmpt;
40.
41.         struct {
42.             ...... //dnnet 的数据,不用理会
43.         } dnports;
44.
45.         __u32       spi;
46.     } uli_u;
47. #define fl_ip_sport uli_u.ports.sport
48. #define fl_ip_dport uli_u.ports.dport
49. #define fl_icmp_type    uli_u.icmpt.type
50. #define fl_icmp_code    uli_u.icmpt.code
51. #define fl_ipsec_spi    uli_u.spi
52. } __attribute__((__aligned__(BITS_PER_LONG/8)));
```

代码段 4-8 flowi 结构

从上面结构定义可以看到，一个数据报文有源、目的地址端口，有 proto 选项，有用户定义的类型，甚至有入接口和出接口，那么，通过这些标识，就可以唯一的确定某用户的业务流。然后你就可以对某一个指定的流查找其路由。好啦，可以这么说，路由是网络内不同业务流的标识，而 flowi 是操作系统内部不同业务流的标识。内核通过从 TCP 或 IP 报文头中抽取相应的信息填入到 flowi 结构中，然后路由查找模块根据这个信息为相应的流找到对应路由。所以说，flowi 就是一个查找 key。

路由的范围放在 flowi 结构的 scope 字段，我们可以想象这个“scope”是到目的地址的距离。它是用来确定如何路由报文和如何归类这些路由。上表的 scope 值用在 fib_result 里的 scope 字段

和 next_hop 结构的 fib_nhs 里的 nh_scope 字段，用户创建的特定路由表应用程序可以定义 scope 的范围是 0~199，当前，Linux Ipv4 经常使用的是 RT_SCOPE_UNIVERSE，RT_SCOPE_LINK 或 RT_SCOPE_HOST。较大的数暗示更接近目的地（除了 RT_SCOPE_NOWHERE，表示目的地不存在）。

每个路由表项即 rtable 的第一部分包含一个目的地址 cache 表项结构，叫 dst_entry，它包扩用来指向管理 cache 表项的相应函数——dst_ops 结构，对于 IP Route Cache 的 dst_ops 值如下（注意表中的路由表项指的是路由 cache 中的表项，不是 FIB 中的表项）：

表格 4-1

dst_ops 里的字段	相应的值或函数	目的
Family	AF_INET	IPv4 地址族.
protocol	ETH_P_IP	链路层的协议字段，必须为 0x0800.
gc	rt_garbage_collect	垃圾回收函数
check	ipv4_dst_check	目前为空函数
destroy	ipv4_dst_destroy	删除路由表项的函数
negative_advice	ipv4_negative_advice	如果任何表项要重定向或者要被删除的时候就调用此函数
link_failure	ipv4_link_failure	发送一个 ICMP unreachable 消息并让这条路由作废，通常此函数由 arp_error_report 调用
update_pmtu	ip_rt_update_pmtu	更新某路由的 MTU 值
entry_size	sizeof(struct rtable)	指定路由表项的大小.

在 raw_sendmsg 函数中调用了 ip_route_output_flow，但它只是简单的调用了 __ip_route_output_key，只是参数有所不同。这些比较重要的参数，要结合上面给出的代码仔细研究。首先，通过检查 dst_entry 的废除字段看是否该路由已被废除。如果是，就调用 ip_rt_put 函数把该路由归还到 slab cache，否则，我们想删除到某目的地址所有的路由。。。我们通过检查 dst_entry 的超时字段看该表项是否过期，或者看 rt_flag 标志是否已被打上 RTCF_REDIRECTED 标志来判断该路由已经被重路由，如果是，就重新计算 32 位 hash 值并调用 rt_del 删除匹配 hash 桶位置的所有路由。

一些路由 cache 函数被直接调用，其中一部分函数被定义成内联函数，比如 ip_rt_put，它通过调用通用的 dst_release 函数删除一条路由。再如 rt_bind_peer，它给某路由创建一条对端表项，从效果上看，它增加了关于到路由表中的目的地址的信息。

当一条新的路由为发送报文准备好后，ip_route_output 调用 rt_intern_hash 函数，rt_inten_hash 用 hash 作为索引到 rt_hash_table 的参数，这个索引只是定位到最匹配的位置，那个位置可以包含 0 个、1 个或多个路由。然后该函数用 rt 中的 flowi 部分去匹配其中的一个路由。如果它找到一个精确匹配的，就把这个路由移到链表的前面，增加它的使用计数器，然后释放新的路由 rt。如果没有匹配的项，那么新路由 rt 会放在链表的前面。如果邻居 cache 好像满了，我们就调用 rt_garbage_collect 函数删除路由，直到有足够的空间放置新路由。如果 rt_intern_hash 不能找到空间，就返回 ENOBUFS 错误代码。

一旦有一条指向非直连主机的外部地址的路由，那么两个主要的路由解析函数：ip_route_output_slow 和 ip_route_input_slow 需要创建一些特别信息。例如，IP 允许设定 MTU，而且 TCP 可以与对端协商最大端长度的选项，那么就可以通过调用 rt_set_nexthop 去设定这些信息。

具体流程就是下面的函数：

现给出__ip_route_output_key 的代码：

```
1.  int __ip_route_output_key(struct rtable **rp, const struct flowi *flp)
2.  {
3.      unsigned hash;
4.      struct rtable *rth;
    为 flp 找到相应的 hash 键值
5.      hash = rt_hash_code(flp->fl4_dst, flp->fl4_src ^ (flp->oif << 5), flp->fl4_tos);
6.      ......
    rt_hash_table 是一个全局变量，根据 hash 值对相应链表进行遍历
7.      for (rth = rt_hash_table[hash].chain; rth; rth = rth->u.rt_next) {
8.          ......
9.          if (rth->fl.fl4_dst == flp->fl4_dst &&
10.             rth->fl.fl4_src == flp->fl4_src &&
11.             rth->fl.iif == 0 &&
12.             rth->fl.oif == flp->oif &&
13.         !((rth->fl.fl4_tos ^ flp->fl4_tos) &(IPTOS_RT_MASK | RTO_ONLINK)))
14.         {
                如果找到了相应的 rt 表项，那么就返回
15.             rth->u.dst.lastuse = jiffies;
16.
17.             rth->u.dst.__use++;
18.         ......
19.             *rp = rth;
20.             return 0;
21.         }
22.
23.     }
24.     ......
    如果没有找到相应表项，表明还没有为该流找到一条路由，于是，将进行路由解析过程
25.     ①return ip_route_output_slow(rp, flp);
26. }
```

代码段 4-9 __ip_route_output_key 函数

①再分析ip_route_output_slow的代码：

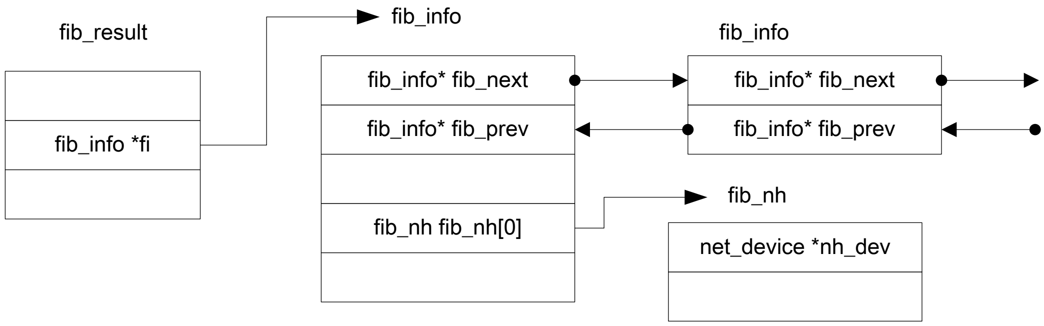
```
1.  /*
2.   * 主要的路由解析函数，在这里的 oldflp 是从 xxx_sendmsg 中传递过来的值，代表了没有找到路由的
3.   * 那条流。
4.   */
5.  static int ip_route_output_slow(struct rtable **rp, const struct flowi *oldflp)
6.  {
7.      u32 tos = RT_FL_TOS(oldflp);
     先把老的 flowi 结构复制到新的 flowi 结构中，并且，还初始化入接口和出接口，注意，入接口
     在此还是 loopback 接口，在只有一个接口的情况下，它等于 2，而老 flowi 结构中的 oif 也许只是 0
8.
9.      struct flowi fl = { .nl_u = { .ip4_u =
10.                     { .daddr = oldflp->fl4_dst,
11.                    .saddr = oldflp->fl4_src,
12.                    .tos = tos & IPTOS_RT_MASK,
                       如果设置了 MSG_DONTROUTE，那么 tos=RTO_ONLINK，于是 scope＝SCOPE_LINK
13.                    .scope = ((tos & RTO_ONLINK) ?
14.                          RT_SCOPE_LINK :
15.                          RT_SCOPE_UNIVERSE),
16.                     } },
17.                .iif = loopback_dev.ifindex,
18.                .oif = oldflp->oif };
19.     struct fib_result res;
20.     unsigned flags = 0;
21.     struct net_device *dev_out = NULL;
22.     int free_res = 0;
23.     int err;
24.
25.     res.fi      = NULL;
```

```
26. #ifdef CONFIG_IP_MULTIPLE_TABLES
27.     res.r       = NULL;
28. #endif
29.
30.     if (oldflp->fl4_src) {
     如果流标识中有源地址，那么就先查找到与源地址对应的地址
31.         err = -EINVAL;
32.         if (MULTICAST(oldflp->fl4_src) ||
33.             BADCLASS(oldflp->fl4_src) ||
34.             ZERONET(oldflp->fl4_src))
35.             goto out;
36.
37.     /* It is equivalent to inet_addr_type(saddr) == RTN_LOCAL */
     此函数内部也调用了fib查找
38.         ②dev_out = ip_dev_find(oldflp->fl4_src);
39.         if (dev_out == NULL)
40.          goto out;
41.
42.     /* I removed check for oif == dev_out->oif here.
43.        It was wrong for two reasons:
44.        1. ip_dev_find(saddr) can return wrong iface, if saddr is
45.           assigned to multiple interfaces.
46.        2. Moreover, we are allowed to send packets with saddr
47.           of another iface. --ANK
48.      */
49.
50.         if (oldflp->oif == 0
51.             && (MULTICAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF)) {
52.          /* Special hack: user can direct multicasts
53.             and limited broadcast via necessary interface
54.             without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
55.             This hack is not just for fun, it allows
56.             vic,vat and friends to work.
57.             They bind socket to loopback, set ttl to zero
58.             and expect that it will work.
59.             From the viewpoint of routing cache they are broken,
60.             because we are not allowed to build multicast path
61.             with loopback source addr (look, routing cache
62.             cannot know, that ttl is zero, so that packet
63.             will not leave this host and route is valid).
64.             Luckily, this hack is good workaround.
65.           */
66.
67.             fl.oif = dev_out->ifindex;
68.             goto make_route;
69.         }
70.
71.         dev_out = NULL;
72.     }
73.
74.     if (oldflp->oif) {
75.         dev_out = dev_get_by_index(oldflp->oif);
76.         err = -ENODEV;
77.         if (dev_out == NULL)
78.          goto out;
79.
80.     /* RACE: Check return value of inet_select_addr instead. */
81.         if (__in_dev_get_rtnl(dev_out) == NULL) {
82.          goto out;   /* Wrong error code */
83.         }
84.
85.         if (LOCAL_MCAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF) {
86.             if (!fl.fl4_src)
87.                 fl.fl4_src = inet_select_addr(dev_out, 0,
88.                               RT_SCOPE_LINK);
89.             goto make_route;
90.         }
91.         if (!fl.fl4_src) {
```

```
92.             if (MULTICAST(oldflp->fl4_dst))
93.          如果目的地址组播地址，走此处，但我们不关心......
94.     }
95.
96.     if (!fl.fl4_dst) {
          当目的地址是 0 的话，那么有必要为源和目的地址填上环回地址，并且接口也是环回接口。
97.         fl.fl4_dst = fl.fl4_src;
98.         if (!fl.fl4_dst)
99.             fl.fl4_dst = fl.fl4_src = htonl(INADDR_LOOPBACK);
100.
101.         dev_out = &loopback_dev;
102.
103.         fl.oif = loopback_dev.ifindex;
104.         res.type = RTN_LOCAL;
105.         flags |= RTCF_LOCAL;
106.         goto make_route;
107.     }
     这里进行路由表查询。注意，所谓的 fib_llokup 就是查找路由表
108.     ③if (fib_lookup(&fl, &res)) {
109.         如果没有在 FIB 表中找到相应路由，那么进入这个分支
110.         res.fi = NULL;
111.         if (oldflp->oif) {
112.             如果为报文指定了输出接口，
113.             /* Apparently, routing tables are wrong. Assume,
114.                that the destination is on link.
115.
116.                WHY? DW.
117.                Because we are allowed to send to iface
118.                even if it has NO routes and NO assigned
119.                addresses. When oif is specified, routing
120.                tables are looked up with only one purpose:
121.                to catch if destination is gatewayed, rather than
122.                direct. Moreover, if MSG_DONTROUTE is set,
123.                we send packet, ignoring both routing tables
124.                and ifaddr state. --ANK
125.
126.
127.                We could make it even if oif is unknown,
128.                likely IPv6, but we do not.
129.              */
130.
131.             if (fl.fl4_src == 0)
132.                 fl.fl4_src = inet_select_addr(dev_out, 0,
133.                                     RT_SCOPE_LINK);
134.             res.type = RTN_UNICAST;
135.             goto make_route;
136.         }
137.
138.         err = -ENETUNREACH;
139.         goto out;
140.     }
141.     free_res = 1;
142.
143.     if (res.type == RTN_LOCAL) {
     此报文的目的地是本机，那么将使用 loopback 设备作为发送载体，而不是用实际设备
144.         if (!fl.fl4_src)
145.             fl.fl4_src = fl.fl4_dst;
146.
147.         dev_out = &loopback_dev;
148.
149.         fl.oif = dev_out->ifindex;
150.
151.         res.fi = NULL;
152.         flags |= RTCF_LOCAL;
153.         goto make_route;
154.     }
155.
156.     #ifdef CONFIG_IP_ROUTE_MULTIPATH
```

```
157.        if (res.fi->fib_nhs > 1 && fl.oif == 0)
158.            fib_select_multipath(&fl, &res);
159.        else
160.    #endif
161.    if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
162.        ④fib_select_default(&fl, &res);
    为本报文流设置源地址，这个值将会赋给 oldflp->fl4_src（在退出此函数以后）
163.    if (!fl.fl4_src)
164.    ⑤  fl.fl4_src = FIB_RES_PREFSRC(res);
    找到了发送接口以及接口索引
165.    dev_out = FIB_RES_DEV(res);
166.    fl.oif = dev_out->ifindex;
167.
  在这里将 fib 查找的结果放入路由 cache 中。不要被 make_route 这样的标签迷惑了。
168.    make_route:
169.    ⑥  err = ip_mkroute_output(rp, &res, &fl, oldflp, dev_out, flags);
170.
171.        if (free_res)
172.            fib_res_put(&res);
173.    out:    return err;
174.    }
```

代码段 4-10 ip_route_output_slow 函数



图表 4-5fib_result、fib_info、fib_nh 的关系

②③先看第一次查找ip_dev_find，它比较简单，只是调用ip_fib_local_table → tb_lookup (ip_fib_local_table, &fl, &res)；它和之后要研究的fib_lookup函数类似，只是不查找ip_fib_main_table：

```
1.  static inline int fib_lookup(const struct flowi *flp, struct fib_result *res)
2.  {
3.      if (ip_fib_local_table->tb_lookup(ip_fib_local_table, flp, res) &&
4.          ip_fib_main_table->tb_lookup(ip_fib_main_table, flp, res))
5.          return -ENETUNREACH;
6.      return 0;
7.  }
```

Fib_lookup 是 FIB 中做路由搜索的主要的前端函数。首先调用 local 表的查找函数，然后再调用本地网的查找函数，如果没有找到就返回 ENETUNREACH，要注意的是 2 个表都必须查找。

那么我们在此要说关于 fib_rules 的一些事情。可以看到在这个版本的 Linux 内核中似乎 fib_rules 没有起作用了，怎么回事？在 Linux2.6.14 中是这么定义的：

static struct fib_rule *fib_rules = &local_rule;

但是在 2.6.18 中已经不是了，在配置内核编译选项的时候如果没有配置 policy_routing，那么

CONFIG_IP_MULTIPLE_TABLES 选项就不会定义，于是 fib_rules 等于 NULL，其相应的初始化函数没有包含在内核中，在整个协议栈中不起作用。而原先版本中 fib_lookup 函数只有一个定义，现在变成 2 个，在定义了 CONFIG_IP_MULTIPLE_TABLES 时，就会调用如下的代码：

```
1.  int fib_lookup(const struct flowi *flp, struct fib_result *res)
2.  {
3.      int err;
4.      struct fib_rule *r, *policy;
5.      struct fib_table *tb;
6.      struct hlist_node *node;
7.
8.      u32 daddr = flp->fl4_dst;
9.      u32 saddr = flp->fl4_src;
10.
11.     hlist_for_each_entry_rcu(r, node, &fib_rules, hlist) {
12.         if (((saddr^r->r_src) & r->r_srcmask) ||
13.             ((daddr^r->r_dst) & r->r_dstmask) ||
14.             (r->r_tos && r->r_tos != flp->fl4_tos) ||
15.              (r->r_ifindex && r->r_ifindex != flp->iif))
16.              continue;
       不管是 main_rule 还是 local_rule，其 action 必定是 RTN_UNICAST
17.         switch (r->r_action) {
18.         case RTN_UNICAST:
19.             policy = r;
20.             break;
21.         case RTN_UNREACHABLE:
22.             return -ENETUNREACH;
23.         default:
24.         case RTN_BLACKHOLE:
25.             return -EINVAL;
26.         case RTN_PROHIBIT:
27.              return -EACCES;
28.         }
    在没有加入新的路由规则时，获取的 tbl 只能是 2 个：ip_fib_local_table 和 ip_fib_main_table
29.         if ((tb = fib_get_table(r->r_table)) == NULL)
30.             continue;
    在这里就和上面的 fib_lookup 的实现有点象了
31.         err = tb->tb_lookup(tb, flp, res);
32.         if (err == 0) {
33.              res->r = policy;
34.              return 0;
35.         }
36.     }
37.     return -ENETUNREACH;
38. }
```

代码段 4-11 fib_lookup 函数

为什么会变化，我想可能有 2 方面原因：第一，代码更清晰了，如果没有配置策略路由，开发人员不必再关心 fib_rules 了，直接处理的就 main 和 local 表；第二，从性能上稍微有提高，因为对于普通的主机，一般不会去配置外部路由协议，也就不会往 fib_rules 加 rules 了，于是查找的时候直接操作 local 和 main 表，省去了遍历 fib_rules 链表的过程。

我们会在 hn_hash_lookup 函数中看到查找正确路由的 3 步曲，这也是普通 hash 表查找过程的 3 部曲，我已分别用序号列出：

```
1.  static int
2.  fn_hash_lookup(struct fib_table *tb, const struct flowi *flp, struct fib_result
    *res)
3.  {
4.      int err;
5.      struct fn_zone *fz;
6.      struct fn_hash *t = (struct fn_hash*)tb->tb_data;
     在介绍 fn_hash{}的时候曾经给读者买过一个关子，fn_zone_list 在这里派上用场了：从掩码最长的
```

```
    zone 开始查找，也就是从最精确的路由 zone 中查找到最匹配的路由。
7.      for (fz = t->fn_zone_list; fz; fz = fz->fz_next)
8.      {
9.          struct hlist_head *head;
10.         struct hlist_node *node;
11.         struct fib_node *f;
        形成查找 fib_node{}的 key
12.         ①u32 k = fz_key(flp->fl4_dst, fz);
        用 key 形成对应 hash 数组的单元
13.         ②head = &fz->fz_hash[fn_hash(k, fz)];
        遍历 fib_node{}链表
14.         ③hlist_for_each_entry(f, node, head, fn_hash) {
15.             if (f->fn_key != k)
16.                 continue;
            当找到同样 key 的节点，用输入的参数填满 res
17.             err = fib_semantic_match(&f->fn_ali
18.                     flp, res,
19.                     f->fn_key, fz->fz_mask,
20.                     fz->fz_order);
21.             if (err <= 0)
22.                 goto out;
23.         }
24.     }
25.     err = 1;
26.     out:
27.
28.     return err;
29. }
```

```
u32 fz_key(u32 dst, struct fn_zone *fz)
{
    return dst & FZ_MASK(fz);
}
```

```
u32 fn_hash(u32 key, struct fn_zone
*fz)
{
    u32 h = ntohl(key)>>(32 -
    fz->fz_order);
    h ^= (h>>20);
    h ^= (h>>10);
    h ^= (h>>5);
    h &= FZ_HASHMASK(fz);
    return h;
}
```

代码段 4-12 fn_hash_lookup 函数

对于路由表的操作，物理操作是直观的和易于理解的。对于表的操作不外乎就是添加、删除、更新等的操作。还有一种操作，是所谓的语义操作，语义操作主要是指诸如计算下一条的地址，把节点转换为路由项，寻找指定信息的路由等。它并不涉及路由表整体框架的理解，而且，函数名也是不言自明的。

```
1.  int fib_semantic_match(struct list_head *head, const struct flowi *flp,
2.          struct fib_result *res, __u32 zone, __u32 mask,
3.          int prefixlen)
4.  {
5.      struct fib_alias *fa;
6.      int nh_sel = 0;
    遍历 fib_alias{}列表
7.      list_for_each_entry_rcu(fa, head, fa_list) {
8.          int err;
9.
10.         if (fa->fa_tos &&
11.             fa->fa_tos != flp->fl4_tos)
12.             continue;
13.
14.         if (fa->fa_scope < flp->fl4_scope)
15.             continue;
16.
17.         fa->fa_state |= FA_S_ACCESSED;
        前面已经说过了 RT_TYPE 和 RTN_SCOPE 之间的关系了
18.         err = fib_props[fa->fa_type].error;
19.         if (err == 0) {
            只有合适的 scope 才能进入
20.             struct fib_info *fi = fa->fa_info;
            在缺省情况下 flag 永远不会是 DEAD 状态，所以只会继续往下走
21.             if (fi->fib_flags & RTNH_F_DEAD)
22.                 continue;
23.
```

```
24.             switch (fa->fa_type) {
25.             case RTN_UNICAST:
26.             case RTN_LOCAL:
27.             case RTN_BROADCAST:
28.             case RTN_ANYCAST:
29.             case RTN_MULTICAST:
            查找吓一跳接口，缺省情况只循环一次
30.                 for_nexthops(fi) {
31.                     if (nh->nh_flags&RTNH_F_DEAD)
32.                         continue;
33.                     if (!flp->oif || flp->oif == nh->nh_oif)
34.                         break;
35.                 }
            缺省情况下，nhsel 肯定小于 1，而其我们已经获得下一跳信息，所以跳出去。
36.                 if (nhsel < 1) {
37.                     goto out_fill_res;
38.                 }
39.                 endfor_nexthops(fi);
40.                 continue;
41.
42.             default:
43.                 printk(KERN_DEBUG "impossible 102\n");
44.                 return -EINVAL;
45.             };
46.         }
47.         return err;
48.     }
49.     return 1;
50.
51. out_fill_res:
52.     res->prefixlen = prefixlen;
53.     res->nh_sel = nh_sel;
54.     res->type = fa->fa_type;
55.     res->scope = fa->fa_scope;
    从 FIB 中获取最重要的下一跳接口信息，要记住往路由 cache 中写下一跳信息就是通过这一次赋值
56.     res->fi = fa->fa_info;
57.
58.     return 0;
59. }
```

代码段 4-13 fib_semantic_match 函数

ip_route_output_slow函数中⑤FIB_RES_PREFSRC看起来其貌不扬，但是它要完成一件比较重要的事，而且要和我们之前讨论配置的时候联系起来。其宏定义如下：

```
#define FIB_RES_PREFSRC(res)   ((res).fi->fib_prefsrc ? : __fib_res_prefsrc(&res))
```

注意，类似于 b = a ? : 10;这样的宏在 VC6 中编译不过，而在 GCC 中可以编译成功，其含义表示如果 a 不等于 0，那么 b＝a；如果 a 等于 0，那么等于 10。

那么在我们前面给接口配置 IP 地址的时候，fib_prefsrc 等于 ifa->ifa_local，这是肯定不等于 NULL，所以 fl.fl4_src＝fib_prefsrc，即接口的本地地址。

__fib_res_prefsrc 直接调 inet_select_addr（FIB_RES_DEV(*res), FIB_RES_GW(*res), res -> scope），

```
1.  u32 inet_select_addr(const struct net_device *dev, u32 dst, int scope)
2.  {
3.      u32 addr = 0;
4.      struct in_device *in_dev;
5.
6.      in_dev = __in_dev_get_rcu(dev);
7.
8.      for_primary_ifa(in_dev) {
```

```
9.          if (ifa->ifa_scope > scope)
10.             continue;
11.         if (!dst || inet_ifa_match(dst, ifa)) {
12.             addr = ifa->ifa_local;
13.             break;
14.         }
15.         if (!addr)
16.             addr = ifa->ifa_local;
17.     } endfor_ifa(in_dev);
18. no_in_dev:
19.
20.     if (addr)
21.         goto out;
22.
23. /* Not loopback addresses on loopback should be preferred
24.    in this case. lo 设备是 dev_base 链表的第一个接口，这点很重要
25.  */
26.
27.     for (dev = dev_base; dev; dev = dev->next) {
28.         if ((in_dev = __in_dev_get_rcu(dev)) == NULL)
29.             continue;
30.
31.         for_primary_ifa(in_dev) {
32.             if (ifa->ifa_scope != RT_SCOPE_LINK &&
33.                 ifa->ifa_scope <= scope) {
34.                  addr = ifa->ifa_local;
35.                  goto out_unlock_both;
36.             }
37.         } endfor_ifa(in_dev);
38.     }
39.     out_unlock_both:
40.
41. out:
42.     return addr;
43. }
```

代码段 4-14 inet_select_addr

4.5.2 当目的地址是远端主机时

ip_route_output_slow函数④，继续往下走，当我们ping的地址不是 127.0.0.1 或本地地址，而是远端主机时，那么下面这个函数相当重用，不过建议读者先跨过这一节，先看看邻居系统是如何工作的。当我们对ping 127.0.0.1 或ping 本地地址的流程熟悉以后，可以再回到这一节。

```
1.  static inline void fib_select_default(const struct flowi *flp, struct fib_result
    *res)
2.  {
3.      if (FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
4.          ip_fib_main_table->tb_select_default(ip_fib_main_table, flp, res);
5.  }
```

代码段 4-15 fib_select_default 函数

```
6.  static void
7.  fn_hash_select_default(struct fib_table *tb, const struct flowi *flp, struct
    fib_result *res)
8.  {
9.      int order, last_idx;
10.     struct hlist_node *node;
11.     struct fib_node *f;
12.     struct fib_info *fi = NULL;
13.     struct fib_info *last_resort;
14.     struct fn_hash *t = (struct fn_hash*)tb->tb_data;
```

```
15.     struct fn_zone *fz = t->fn_zones[0];
16.
17.     if (fz == NULL)
18.         return;
19.
20.     last_idx = -1;
21.     last_resort = NULL;
22.     order = -1;
23.
24.     hlist_for_each_entry(f, node, &fz->fz_hash[0], fn_hash) {
25.         struct fib_alias *fa;
26.
27.         list_for_each_entry(fa, &f->fn_alias, fa_list) {
28.             struct fib_info *next_fi = fa->fa_info;
29.
30.             if (fa->fa_scope != res->scope ||
31.                fa->fa_type != RTN_UNICAST)
32.                 continue;
33.
34.             if (next_fi->fib_priority > res->fi->fib_priority)
35.                 break;
36.             if (!next_fi->fib_nh[0].nh_gw ||
37.                next_fi->fib_nh[0].nh_scope != RT_SCOPE_LINK)
38.                 continue;
39.             fa->fa_state |= FA_S_ACCESSED;
40.
41.             if (fi == NULL) {
42.                 if (next_fi != res->fi)
43.                     break;
44.             } else if (!fib_detect_death(fi, order, &last_resort,
45.                          &last_idx, &fn_hash_last_dflt)) {
46.                 res->fi = fi;
47.
48.                 fn_hash_last_dflt = order;
49.                 goto out;
50.             }
51.             fi = next_fi;
52.             order++;
53.         }
54.     }
55.
56.     if (order <= 0 || fi == NULL) {
57.         fn_hash_last_dflt = -1;
58.         goto out;
59.     }
60.
61.     if(!fib_detect_death(fi, order, &last_resort,&last_idx,&fn_hash_last_dflt))
62.     {
63.         res->fi = fi;
64.         fn_hash_last_dflt = order;
65.         goto out;
66.     }
67.
68.     if (last_idx >= 0) {
69.         res->fi = last_resort;
70.      }
71.     fn_hash_last_dflt = last_idx;
72.     out:
73. }
```

代码段 4-16 fn_hash_select_default 函数

4.5.3 创建对应路由 cache 表项

⑥ip_mkroute_output函数又直接调用ip_mkroute_output_def：

```
1.  static inline int ip_mkroute_output_def(struct rtable **rp,
2.                  struct fib_result* res,
```

```
3.                  const struct flowi *fl,
4.                  const struct flowi *oldflp,
5.                  struct net_device *dev_out,
6.                  unsigned flags)
7.  {
8.      struct rtable *rth = NULL;
9.      int err = __mkroute_output(&rth, res, fl, oldflp, dev_out, flags);
10.     unsigned hash;
11.     if (err == 0) {
12.         hash = rt_hash_code(oldflp->fl4_dst,
13.                     oldflp->fl4_src ^ (oldflp->oif << 5));
14.         err = rt_intern_hash(hash, rth, rp);
15.     }
16.
17.     return err;
18. }
```

代码段 4-17 ip_mkroute_output_def 函数

__mkroute_output 是一个很重要的函数，它指定输出的函数——ip_output。

```
1.  static inline int __mkroute_output(struct rtable **result,
2.                 struct fib_result* res,
3.                 const struct flowi *fl,
4.                 const struct flowi *oldflp,
5.                 struct net_device *dev_out,
6.                 unsigned flags)
7.  {
8.      struct rtable *rth;
9.      struct in_device *in_dev;
10.     u32 tos = RT_FL_TOS(oldflp);
11.     int err = 0;
12.
13.     if (LOOPBACK(fl->fl4_src) && !(dev_out->flags&IFF_LOOPBACK))
14.         return -EINVAL;
15.
16.     if (fl->fl4_dst == 0xFFFFFFFF)
17.         res->type = RTN_BROADCAST;
18.     else if (MULTICAST(fl->fl4_dst))
19.         res->type = RTN_MULTICAST;
20.     else if (BADCLASS(fl->fl4_dst) || ZERONET(fl->fl4_dst))
21.         return -EINVAL;
22.
23.     if (dev_out->flags & IFF_LOOPBACK)
24.         flags |= RTCF_LOCAL;
25.
26.     /* get work reference to inet device */
27.     in_dev = in_dev_get(dev_out);
28.     if (res->type == RTN_BROADCAST) {
29.         flags |= RTCF_BROADCAST | RTCF_LOCAL;
30.         if (res->fi) {
31.             res->fi = NULL;
32.         }
33.     } else if (res->type == RTN_MULTICAST) {
34.         flags |= RTCF_MULTICAST|RTCF_LOCAL;
35.         if (!ip_check_mc(in_dev, oldflp->fl4_dst, oldflp->fl4_src,
36.                  oldflp->proto))
37.             flags &= ~RTCF_LOCAL;
38.         /* If multicast route do not exist use
39.            default one, but do not gateway in this case.
40.            Yes, it is hack.
41.          */
42.         if (res->fi && res->prefixlen < 4) {
43.             res->fi = NULL;
44.         }
45.     }
46.
        创建一个 rtable{}结构，并且初始化
47.     rth = dst_alloc(&ipv4_dst_ops);
```

```
48.
49.     rth->u.dst.flags= DST_HOST;
50.     if (in_dev->cnf.no_xfrm)
51.         rth->u.dst.flags |= DST_NOXFRM;
52.     if (in_dev->cnf.no_policy)
53.         rth->u.dst.flags |= DST_NOPOLICY;
54.
55.     rth->fl.fl4_dst = oldflp->fl4_dst;
56.     rth->fl.fl4_tos = tos;
57.     rth->fl.fl4_src = oldflp->fl4_src;
58.     rth->fl.oif = oldflp->oif;
59.     rth->rt_dst = fl->fl4_dst;
60.     rth->rt_src = fl->fl4_src;
61.     rth->rt_iif = oldflp->oif ? : dev_out->ifindex;
62.     /* get references to the devices that are to be hold by the routing
63.        cache entry */
64.     rth->u.dst.dev  = dev_out;
65.
66.     rth->idev   = in_dev_get(dev_out);
67.     rth->rt_gateway = fl->fl4_dst;
68.     rth->rt_spec_dst= fl->fl4_src;
69.
70.     rth->u.dst.output=ip_output;
71.
72.     RT_CACHE_STAT_INC(out_slow_tot);
```

如果 type 是 RTN_LOCAL，那么设置目的 cache 的 input 字段为 ip_local_deliver，于是此包发给传输层协议的。而且，如果该路由的路由表项被设置为 RTCF_LOCAL 标志，那么我们知道这个包一定是本地路由。

```
73.     if (flags & RTCF_LOCAL) {
74.         rth->u.dst.input = ip_local_deliver;
75.         rth->rt_spec_dst = fl->fl4_dst;
76.     }
77.     if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
78.         rth->rt_spec_dst = fl->fl4_src;
79.         if (flags & RTCF_LOCAL &&
80.             !(dev_out->flags & IFF_LOOPBACK)) {
81.             rth->u.dst.output = ip_mc_output;
82.             RT_CACHE_STAT_INC(out_slow_mc);
83.         }
84.     }
85.
86.     rt_set_nexthop(rth, res, 0);
87.
88.     rth->rt_flags = flags;
89.
90.     *result = rth;
91.     cleanup:
92.
93.     return err;
94. }
```

代码段 4-18 __mkroute_output 函数

当 rtable 结构被创建之后，它应该被放入全局链表中了，现在我们回过头再看看 rt_intern_hash：

```
1.  static int rt_intern_hash(unsigned hash, struct rtable *rt, struct rtable **rp)
2.  {
3.      struct rtable   *rth, **rthp;
4.      unsigned long   now;
5.      struct rtable *cand, **candp;
6.      u32     min_score;
7.      int     chain_length;
8.      int attempts = !in_softirq();
9.
10. restart:
11.     chain_length = 0;
12.     min_score = ~(u32)0;
13.     cand = NULL;
14.     candp = NULL;
```

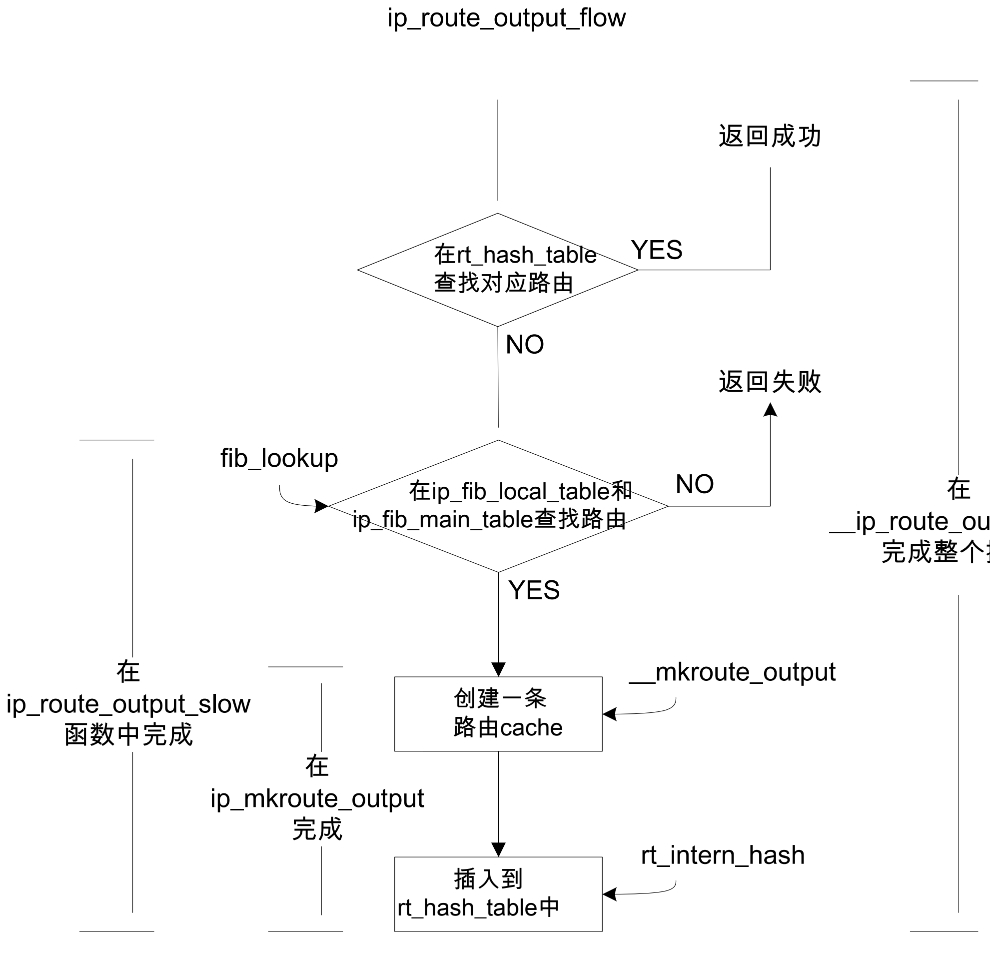
```
15.     now = jiffies;
    这就是我们开始介绍的路由表 cache 的全局变量
16.     rthp = &rt_hash_table[hash].chain;
17.
18.     while ((rth = *rthp) != NULL) {
19. #ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
20.         if (!(rth->u.dst.flags & DST_BALANCED) &&
21.             compare_keys(&rth->fl, &rt->fl)) {
22. #else
23.             if (compare_keys(&rth->fl, &rt->fl)) {
24. #endif
25.             /* 获得第一个节点 */
26.              *rthp = rth->u.rt_next;
27.
             由于查找没有上锁，这个删除动作必须对在插入到 hash 链表头的操作可见，所以采用读写锁
             进行保护
28.              rcu_assign_pointer(rth->u.rt_next,
29.              rt_hash_table[hash].chain);
30.
31.              rcu_assign_pointer(rt_hash_table[hash].chain, rth);
32.
33.              rth->u.dst.__use++;
34.
35.              rth->u.dst.lastuse = now;
36.
37.              rt_drop(rt);
38.              *rp = rth;
39.              return 0;
40.         }
41.
42.         if (!atomic_read(&rth->u.dst.__refcnt)) {
43.             u32 score = rt_score(rth);
44.
45.             if (score <= min_score) {
46.                 cand = rth;
47.                 candp = rthp;
48.                 min_score = score;
49.             }
50.         }
51.         chain_length++;
52.         rthp = &rth->u.rt_next;
53.     }
54.
55.     if (cand) {
56.      /* ip_rt_gc_elasticity used to be average length of chain
57.       * length, when exceeded gc becomes really aggressive.
58.       *
59.       * The second limit is less certain. At the moment it allows
60.       * only 2 entries per bucket. We will see.
61.       */
62.         if (chain_length > ip_rt_gc_elasticity) {
63.             *candp = cand->u.rt_next;
64.             rt_free(cand);
65.         }
66.     }
67.
68. /* 尝试把路由绑定到 arp 上，这只有在此路由是出口路由或者是单播转发路径时才能这样处理
69.  */
70.     if (rt->rt_type == RTN_UNICAST || rt->fl.iif == 0) {
71.         int err = arp_bind_neighbour(&rt->u.dst);
72.         if (err) {
73.              ......
74.             return -ENOBUFS;
75.         }
76.     }
77.
78.     rt->u.rt_next = rt_hash_table[hash].chain;
79.     rt_hash_table[hash].chain = rt;
80.
```

```
81.     *rp = rt;
82.     return 0;
83. }
```

代码段 4-19 rt_intern_hash 函数

好了，到了总结一番的时候了：

1) Linux 内核没有名字叫做路由表/route table 的表，不要被 rtable 迷惑了，它不是存放真正路由的地方，它是 cache。FIB 表才是值得叫做路由表的东西。
2) FIB 存放所有的路由信息，只有当要发送报文的时候（也许在接受报文的时候也是这样的）才将已经查过的路由信息放入 route cache 中，在没有进行数据通信之前，cache 中是没有数据的。那么结合__ip_route_output_key 的思路可以得出下面的逻辑：



图表 4-6 __ip_route_output_key 内部逻辑和 FIB、路由 cache 之间的关系

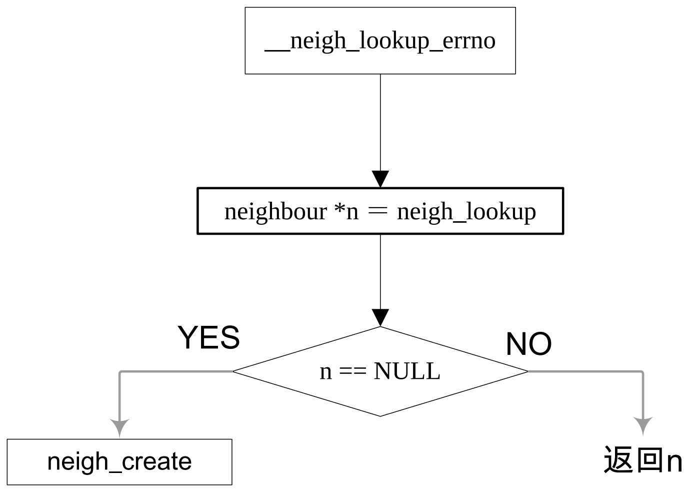
路由查询结果还不能直接供发送IP数据报使用，接下来，还必须根据这个查询结果生成一个路由目的入口(dst_entry)，根据目的入口才可以发送IP数据报，目的入口用结构体struct dst_entry表示，在实际使用时，还在它的外面包装了一层，形成一个结构体struct rtable。

在研究rt_intern_hash函数时发现，在把新路由放在链表之前，我们要调用arp_bind_neighbour使之能够绑定到一个邻居cache。这是什么意思呢？

4.5.4 创建对应邻居表项

```
1.  int arp_bind_neighbour(struct dst_entry *dst)
2.  {
3.      struct net_device *dev = dst->dev;
4.      struct neighbour *n = dst->neighbour;
5.
6.      if (dev == NULL)
7.          return -EINVAL;
8.      if (n == NULL) {
9.          u32 nexthop = ((struct rtable*)dst)->rt_gateway;
        如果是环回设备或点到点设备接口，下一跳是 0
10.         if (dev->flags&(IFF_LOOPBACK|IFF_POINTOPOINT))
11.             nexthop = 0;
        我们把下一跳地址作为创建邻居的 key 值
12.         n = __neigh_lookup_errno(&arp_tbl, &nexthop, dev);
13.
14.         dst->neighbour = n;
15.     }
16.     return 0;
17. }
```

代码段 4-20arp_bind_neighbour 函数



图表 4-7__neigh_lookup_errno 内部逻辑图

在这里 neigh_lookup 很简单，只是在 arp_tbl 中查找相应表现，查找算法依然是我们常见的 hash 算法，我们就不用在多说了。我们重点说一下 neigh_create 函数：

```
1.  struct neighbour *neigh_create(struct neigh_table *tbl, const void *pkey,
2.                 struct net_device *dev)
3.  {
4.      u32 hash_val;
5.      int key_len = tbl->key_len;
6.      int error;
    不仅创建 neighbour 结构，而且还挂上了定时器，将来定期会执行 neigh_timer_handler。
    还要注意 neighbour→output 此时被设置为 neigh_blackhole，即当还没有完全初始化的时候，所有
    的报文都不会发送出去，而是简单的将报文释放
7.      struct neighbour *n1, *rc, *n = neigh_alloc(tbl);
8.
9.      memcpy(n->primary_key, pkey, key_len);
10.     neigh 的 dev 指向目前的设备
11.     n->dev = dev;
12.
13.
14. /* 和协议相关的设置 */
    调用的是 arp_constructor，在 arp_tbl 中定义，下面我们会介绍
15.     if (tbl->constructor && (error = tbl->constructor(n)) < 0) {
```

```
16.         rc = ERR_PTR(error);
17.         goto out_neigh_release;
18.     }
19.
20. /* Device specific setup. */
    这个 parm 实际是 in_device 的 arp_parms，在上面的 constructor 中指定，但是在以太网的情况下
    neigh_setup 是 NULL
21.     if (n->parms->neigh_setup &&
22.        (error = n->parms->neigh_setup(n)) < 0) {
23.      ......
24.     }
25.
26.     n->confirmed = jiffies - (n->parms->base_reachable_time << 1);
27.
28.     if (atomic_read(&tbl->entries) > (tbl->hash_mask + 1))
29.         neigh_hash_grow(tbl, (tbl->hash_mask + 1) << 1);
30.
31.     hash_val = tbl->hash(pkey, dev) & tbl->hash_mask;
32.
33.     if (n->parms->dead) {
34.         rc = ERR_PTR(-EINVAL);
35.         goto out_tbl_unlock;
36.     }
37.
38.     for (n1 = tbl->hash_buckets[hash_val]; n1; n1 = n1->next) {
39.         if (dev == n1->dev && !memcmp(n1->primary_key, pkey, key_len))
40.        {
41.             rc = n1;
42.             goto out_tbl_unlock;
43.         }
44.     }
45.
46.     n->next = tbl->hash_buckets[hash_val];
47.     tbl->hash_buckets[hash_val] = n;
48.     n->dead = 0;
49.
    创建了一个邻居，并把它加入 hash 桶中
50.     rc = n;
51. out:
52.     return rc;
53.     out_tbl_unlock:
54.
55. out_neigh_release:
56.     neigh_release(n);
57.     goto out;
58. }
```

代码段 4-21 neigh_create 函数

大家不要误以为 neigh_alloc 只是一个简单的分配内存的函数，实际上它是进行后续研究的最重用的纽带，因为它分配的一个数据结构和一个定时器回调函数决定了整个 2 层系统的运作，如果不知道这一特点，将会把自己迷失在繁杂的代码丛林中。

```
1.  static struct neighbour *neigh_alloc(struct neigh_table *tbl)
2.  {
3.      struct neighbour *n = NULL;
4.      unsigned long now = jiffies;
5.      int entries;
6.
7.      entries = atomic_inc_return(&tbl->entries) - 1;
8.      if (entries >= tbl->gc_thresh3 ||
9.         (entries >= tbl->gc_thresh2 &&
10.         time_after(now, tbl->last_flush + 5 * HZ))) {
11.         if (!neigh_forced_gc(tbl) &&
12.            entries >= tbl->gc_thresh3)
13.             goto out_entries;
14.     }
```

```
15.
16.     n = kmem_cache_alloc(tbl->kmem_cachep, SLAB_ATOMIC);
17.
18. memset(n, 0, tbl->entry_size);
     初始化 arp 的队列，这个队列是专门用于发送 arp 报文的
19.     skb_queue_head_init(&n->arp_queue);
20.
21.     n->updated    = n->used = now;
22.     n->nud_state  = NUD_NONE;
23.     n->output     = neigh_blackhole;
24.     n->parms  = neigh_parms_clone(&tbl->parms);
25.     init_timer(&n->timer);
26.     n->timer.function = neigh_timer_handler;
27.     n->timer.data     = (unsigned long)n;
    指向 arp_tbl
28.     n->tbl        = tbl;
29.     atomic_set(&n->refcnt, 1);
30.     n->dead       = 1;
31. out:
32.     return n;
33.
34. out_entries:
35.     atomic_dec(&tbl->entries);
36.     goto out;
37. }
```

代码段 4-22 neigh_alloc 函数

现在来看看 arp_constructor 函数：

```
1.  static int arp_constructor(struct neighbour *neigh)
2.  {
3.      u32 addr = *(u32*)neigh->primary_key;
4.      struct net_device *dev = neigh->dev;
5.      struct in_device *in_dev;
6.      struct neigh_parms *parms;
7.
8.      neigh->type = inet_addr_type(addr);
9.
10.     rcu_read_lock();
11.     in_dev = __in_dev_get_rcu(dev);
12.     .....
13.
14.     parms = in_dev->arp_parms;
15.     __neigh_parms_put(neigh->parms);
16.     neigh->parms = neigh_parms_clone(parms);
17.
18.     if (dev->hard_header == NULL) {
19.         neigh->nud_state = NUD_NOARP;
20.         neigh->ops = &arp_direct_ops;
21.         neigh->output = neigh->ops->queue_xmit;
22.     } else {
23.      /* Good devices 不仅仅有以太网的设备
24.             ARPHRD_ETHER: (ethernet, apfddi)
25.             ARPHRD_FDDI: (fddi)
26.             ARPHRD_IEEE802: (tr)
27.             ARPHRD_METRICOM: (strip)
28.             ARPHRD_ARCNET:
29.           等等，目前还没有实现
30.       */
31.
32. #if 1
33.     switch (dev->type) {
34.     default:
35.         break;
36.     ......
37.     }
38. #endif
39.     if (neigh->type == RTN_MULTICAST) {
```

```
static struct neigh_ops arp_direct_ops
= {
.family =       AF_INET,
.output =       dev_queue_xmit,
.connected_output =dev_queue_xmit,
.hh_output =        dev_queue_xmit,
.queue_xmit =       dev_queue_xmit,
};
```

```
static struct neigh_ops arp_hh_ops = {
.family =       AF_INET,
.solicit =      arp_solicit,
.output =       neigh_resolve_output,
.connected_output =neigh_resolve_output,
.hh_output =        dev_queue_xmit,
.queue_xmit =       dev_queue_xmit,
};
```

```
40.         neigh->nud_state = NUD_NOARP;
41.         arp_mc_map(addr, neigh->ha, dev, 1);
42.     } else if (dev->flags&(IFF_NOARP|IFF_LOOPBACK)) {
     loopback 属于 NOARP 状态，
43.         neigh->nud_state = NUD_NOARP;
44.         memcpy(neigh->ha, dev->dev_addr, dev->addr_len);
45.     } else if (neigh->type == RTN_BROADCAST || dev->flags&IFF_POINTOPOINT) {
46.         neigh->nud_state = NUD_NOARP;
47.         memcpy(neigh->ha, dev->broadcast, dev->addr_len);
48.     }
    如果以太网驱动调用 ether_setup 去初始化 net_device，那么该函数指针指向
    eth_header_cache，所以，ops 都指向了 arp_hh_ops。
49.     if (dev->hard_header_cache)
50.         neigh->ops = &arp_hh_ops;
51.     else
52.         neigh->ops = &arp_generic_ops;
     实际上不管 neigh 的状态，就把 neigh→output 指向 neigh_resolve_output 函数
53.     if (neigh->nud_state&NUD_VALID)
54.         neigh->output = neigh->ops->connected_output;
55.     else
56.         neigh->output = neigh->ops->output;
57.     }
58.     return 0;
59. }
```

```
static struct neigh_ops arp_direct_ops = {
.family =           AF_INET,
.output =           dev_queue_xmit,
.connected_output = dev_queue_xmit,
.hh_output =        dev_queue_xmit,
.queue_xmit =       dev_queue_xmit,
};
```

代码段 4-23 arp_constructor 函数

这里看到两个特殊的函数指针，分别是 hard_header 和 hard_header_cache，它们的作用是什么呢？见下文：

hard_header，该成员被以太网设备驱动程序用于为每一个待发送数据报构建以太网首部，系统中所有以太网设备驱动程序共享一个函数即 eth_header。所有数据报在传递给该函数之前，其 skb 头部预留了以太网首部的空间，data 成员指向网络层首部，eth_header 将 data 成员指向以太网首部，并为以太网首部填入目的以太网地址，源以太网地址和网络层协议类型(ETH_P_IP 或 ETH_P_ARP)，该函数在协议栈中主要有两处被用到，一是 ARP 模块中，发送 ARP 请求或应答前，构建以太网首部用；另一处是在 IP 数据发送过程中，构建以太网首部用。

hard_header_cache，用于创建以太网首部的高速缓冲，每一个邻居节点都有一个 struct hh_cache *hh 成员，用于缓冲该邻居节点的以太网首部，有了这个缓冲，以后再向这个邻居发数 IP 数据的时候，不必再调用 hard_header 构建以太网首部，而是直接从 hh 中拷贝即可。

我们在 arp_constructor 函数中指定了三个全局变量，分别定义在 arp.c 中，记住：第一次输出用 neigh_reslove_output 函数，后面的用 dev_queue_xmit 函数。在 neighbour{}结构中有一个 net_device{}结构变量，是这个邻居和本机连接起来的网络接口设备指针。此结构中的 output 函数指针是往这个邻居发送数据的函数，它初始化为和该邻居相连接的接口函数的发送函数。在 neigh_ops 函数中有两个相似的 output 函数：output 和 connected_output，前者用于一般情况下的发送过程，后者是已经建立连接，保证该邻居在这段时间内不会出现位置改变时调用的函数指针。在连接尚未建立时，neighbour→output 初始化为 neigh_ops→output，在连接建立后，neighbour→output 变成 neigh_ops→connected_output 成员。参考 neigh_connect( )和 neigh_suspect( )。

4.6 回到发送的路径

4.6.1.1. IP 层发送过程

扎了一个很深的猛子，不知道读者回过神来没。但是我们还得继续我们的发送之旅。读者要问了：搞了这么久，怎么还没把一个报文发出去呢？不急，马上就到了。

5.4.1 节我们提到了查找路由，当找到一条合适的路由之后，我们就从 ip_route_output_flow 中返回了，继续执行我们的发送报文过程，在判断 inet->hdrincl＝＝0 之后，我们就开始发送报文了，就是我们之前提到的 ip_push_pending_frames:

```
1.  /*
2.   *  把某 socket 之上所有未处理的 IP 分片组合成一个 IP 报文，然后把它们发送出去
3.   */
4.  int ip_push_pending_frames(struct sock *sk)
5.  {
6.      struct sk_buff *skb, *tmp_skb;
7.      struct sk_buff **tail_skb;
8.      struct inet_sock *inet = inet_sk(sk);
9.      struct ip_options *opt = NULL;
10.     struct rtable *rt = inet->cork.rt;
11.     struct iphdr *iph;
12.     __be16 df = 0;
13.     __u8 ttl;
14.     int err = 0;
    把 skb 从发送队列中取出
15.     if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
16.         goto out;
17.     tail_skb = &(skb_shinfo(skb)->frag_list);
18. 
19. /* move skb->data to ip header from ext header */
    把 data 指针往 ip 头挪
20.     if (skb->data < skb->nh.raw)
21.         __skb_pull(skb, skb->nh.raw - skb->data);
22.     while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue)) != NULL) {
23.         __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
24.         *tail_skb = tmp_skb;
25.         tail_skb = &(tmp_skb->next);
26.         skb->len += tmp_skb->len;
27.         skb->data_len += tmp_skb->len;
28.         skb->truesize += tmp_skb->truesize;
29.         __sock_put(tmp_skb->sk);
30.         tmp_skb->destructor = NULL;
31.         tmp_skb->sk = NULL;
32.     }
33. 
34. /* Unless user demanded real pmtu discovery (IP_PMTUDISC_DO), we allow
35.  * to fragment the frame generated here. No matter, what transforms
36.  * how transforms change size of the packet, it will come out.
37.  */
38.     if (inet->pmtudisc != IP_PMTUDISC_DO)
39.         skb->local_df = 1;
40. 
41. /* DF bit is set when we want to see DF on outgoing frames.
42.  * If local_df is set too, we still allow to fragment this frame
43.  * locally. */
44.     if (inet->pmtudisc == IP_PMTUDISC_DO ||
45.        (skb->len <= dst_mtu(&rt->u.dst) &&
46.         ip_dont_fragment(sk, &rt->u.dst)))
47.         df = htons(IP_DF);
48. 
49.     if (inet->cork.flags & IPCORK_OPT)
50.         opt = inet->cork.opt;
51. 
52.     if (rt->rt_type == RTN_MULTICAST)
53.         ttl = inet->mc_ttl;
```

```
54.     else
55.         ttl = ip_select_ttl(inet, &rt->u.dst);
    对 IP 头进行设置，记住我们刚才已经把 data 指向了 ip 头
56.     iph = (struct iphdr *)skb->data;
57.     iph->version = 4;
58.     iph->ihl = 5;
59.     if (opt) {
60.         iph->ihl += opt->optlen>>2;
61.         ip_options_build(skb, opt, inet->cork.addr, rt, 0);
62.     }
63.     iph->tos = inet->tos;
64.     iph->tot_len = htons(skb->len);
65.     iph->frag_off = df;
66.     ip_select_ident(iph, &rt->u.dst, sk);
67.     iph->ttl = ttl;
68.     iph->protocol = sk->sk_protocol;
69.     iph->saddr = rt->rt_src;
70.     iph->daddr = rt->rt_dst;
71.     ip_send_check(iph);
72.
73.     skb->priority = sk->sk_priority;
    这句话是重中之重！我们之前查找路由的努力，就在这里派上用场： skb 中一个 dst 成员指向了路由表
    cache 中的 dst 成员，该 dst 的__refcnt 引用计数加一
74.     skb->dst = dst_clone(&rt->u.dst);
75.
76. /* Netfilter gets whole the not fragmented skb. */
    由于我们不关心 IP Filter 技术，所以，请无视 NF_HOOK 这个宏，读者请把下面这行代码看成：
    dst_output(skb)就可以了，其他参数不用细究。
77.     err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
78.           skb->dst->dev, dst_output);
79. ......
80.
81. out:
82.     inet->cork.flags &= ~IPCORK_OPT;
83.     kfree(inet->cork.opt);
84.     inet->cork.opt = NULL;
85.     if (inet->cork.rt) {
86.         ip_rt_put(inet->cork.rt);
87.         inet->cork.rt = NULL;
88.     }
89.     return err;
90.
91. error:
92.    goto out;
93. }
```

代码段 4-24 ip_push_pending_frames 函数

除了 ping 应用程序以外，我们还可以用 IPPROTO_RAW 参数来使用 RAW 接口，比如 OSPF 的组播报文发送以及 RSVP 等协议报文的发送，都使用 socket(AF_INET, SOCK_RAW, IPPROTO_OSPF 或 IPPROTO_RSVP)这样的代码，故 ip_push_pending_frames 是比较有研究的意义的。

而其他使用 IPPROTO_RAW 作为 socket 系统调用最后一个参数的应用程序，内核中对应的发送函数就是 raw_send_hdrinc：

```
1.  static int raw_send_hdrinc(struct sock *sk, void *from, size_t length,
2.          struct rtable *rt,
3.          unsigned int flags)
4.  {
5.      struct inet_sock *inet = inet_sk(sk);
6.      int hh_len;
7.      struct iphdr *iph;
8.      struct sk_buff *skb;
9.      int err;
```

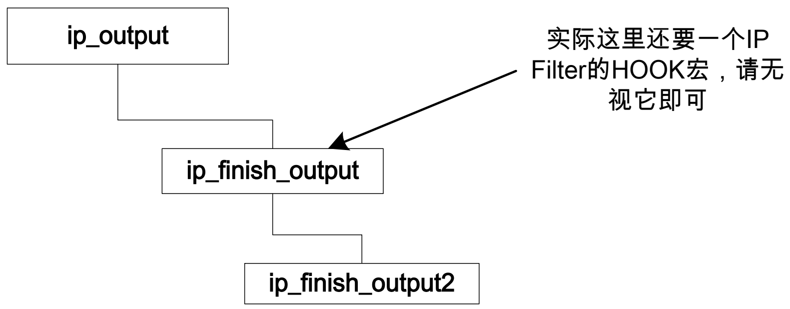
```
     发送长度大于 MTU 的情况必须退出
10.      if (length > rt->u.dst.dev->mtu) {
11.          ......
12.          return -EMSGSIZE;
13.      }
14.      if (flags&MSG_PROBE)
15.          goto out;
16.
17.      hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
     这里和之前的 ip_push_pending_frames 不同，因为报文没有放在发送队列，所以此时才创建 skb
18.      skb = sock_alloc_send_skb(sk, length+hh_len+15,
19.                  flags&MSG_DONTWAIT, &err);
20.
21.      skb_reserve(skb, hh_len);
22.
23.      skb->priority = sk->sk_priority;
     和上面的发送函数一样，也要把 skb→dst 指向 rt→u.dst，很重要的一个过程
24.      skb->dst = dst_clone(&rt->u.dst);
     把 iph 指针指向 skb 的 ip 头位置
25.      skb->nh.iph = iph = (struct iphdr *)skb_put(skb, length);
26.
27.      skb->ip_summed = CHECKSUM_NONE;
28.
29.      skb->h.raw = skb->nh.raw;
     实际上把用户层的发送数据拷贝到 skb 指向的 iph 起始位置
30.      err = memcpy_fromiovecend((void *)iph, from, 0, length);
31.      ......
32.
33.      /* We don't modify invalid header */
34.      if (length >= sizeof(*iph) && iph->ihl * 4U <= length) {
35.          if (!iph->saddr)
36.              iph->saddr = rt->rt_src;
37.          iph->check   = 0;
38.          iph->tot_len = htons(length);
39.          if (!iph->id)
40.              ip_select_ident(iph, &rt->u.dst, NULL);
41.
42.          iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
43.      }
    同样经过 NF_HOOK 来调用 dst_output 函数：
44.      err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
45.               dst_output);

     ......
46.      kfree_skb(skb);
47.      error:
48.      IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
49.      return err;
50. }
```

代码段 4-25 raw_send_hdrinc 函数

此函数和上面那个 ip_push_pending_frames 没有太多区别，所以我们继续往下研究。

dst_output 函数直接调用 skb->dst->output，dst 是上面第 27 行赋值的，那 dst→output 函数是哪个呢？请回到__mkroute_output 函数，它就对 dst→output 函数进行了赋值：ip_output



图表 4-8ip_output 函数调用树

ip_output 函数其实很简单，只是做了 skb->dev = dev; skb->protocol = ETH_P_IP;这么两个赋值操作后就调用 ip_finish_output 函数。后者几乎没有任何变化地调用 ip_finish_output2 函数。那么 ip_finish_output2 函数我们要研究研究：

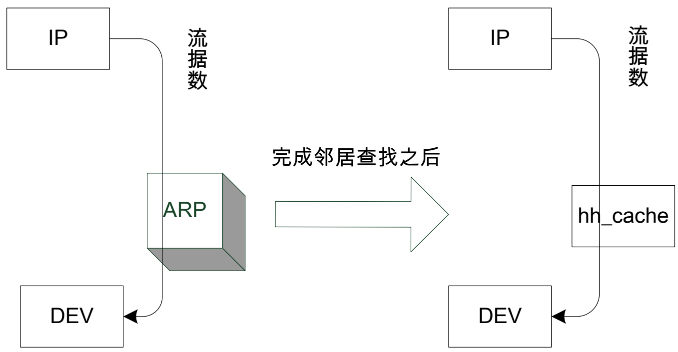
```
1.  static inline int ip_finish_output2(struct sk_buff *skb)
2.  {
3.      struct dst_entry *dst = skb->dst;
4.      struct hh_cache *hh = dst->hh;
5.      struct net_device *dev = dst->dev;
6.      int hh_len = LL_RESERVED_SPACE(dev);
7.
       ......
       第一次进入到这里的时候，hh 等于 NULL，所以执行 neighbour→output 函数，它实际指向
       neigh_resolve_output，请读者回顾 arp_constructor 函数，我们是在那个函数中初始化
       neighbour 的，下一节我们接收 neigh_resolve_output 函数
8.      if (hh) {
9.          int hh_alen;
        hh_alen 等于 16，而 hh→hh_len 等于 14
10.         hh_alen = HH_DATA_ALIGN(hh->hh_len);
            把 2 层报文头一次性拷贝到 skb 中，我们会在下一节中
11.         memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
12.
13.         skb_push(skb, hh->hh_len);
     那这里的 hh_output 是什么呢？这里我们先告诉大家，它指向了 dev_queue_xmit,它是如何指向的我
     们将在下面的 neigh_hh_init 中介绍。
14.         return hh->hh_output(skb);
15.     }
16.     else if (dst->neighbour)
17.         return dst->neighbour->output(skb);
18.
19.     return -EINVAL;
20. }
```

代码段 4-26 ip_finish_output2

原以为要发送出去了，没想到里面还有这么多名堂，那个 hh 到底是什么东西？难道 neighbour{}不负责底层的数据发送吗？

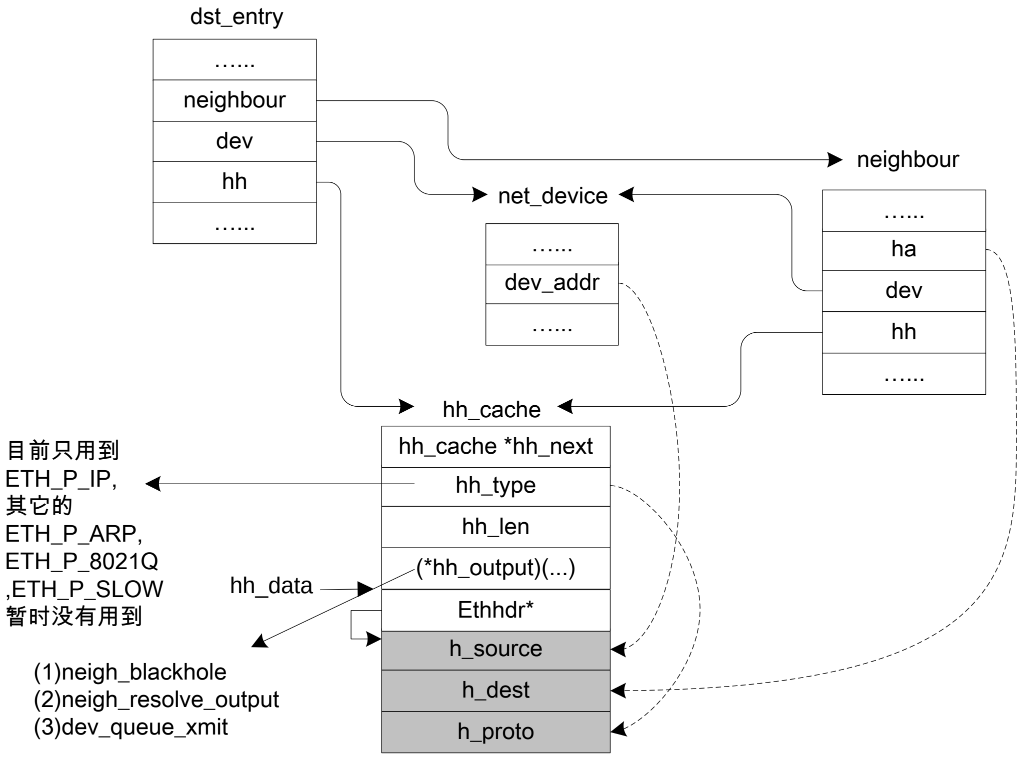
4.6.1.2. 揭密 hh_cache

Linux 内核中有一个数据结构叫 hh_cache{}，代码中没有详细的注释，初看这个数据结构会不知其所用，因为 hh 是什么的缩写也不甚清楚。好了，让我告诉你吧，hh 就是 hard header 的缩写，和 cache 连在一起就是"hard header cache"的意思。我知道读者们还是没有明白，那么先看下面一副图：



虽然我们下一节才开始介绍 ARP，但是我们还是可以从上图中推断出 hh_cache 的作用：硬编址高缓，就是存放邻居 L2 层接口地址和本机 L2 层接口地址以及协议号的缓存。正好这个缓存可以立即拷贝到 sk_buff{}→mac 头中，而不用经过 ARP 协议。上图中 ARP 的框画比较厚重，而 hh_cache 比较“薄”是有意而为之，因为从下一节可以看到 ARP 比较复杂，而 hh 由于保存了必要的信息，使报文的发送变得“轻盈”起来，所以，hh_cache 就是 ARP 缓存的一部分。唉，好累呀。转了这么大个弯。

下图显示了 hh_cache 和其他各部分之间的关系。



图表 4-9 hh_cache 的结构关系图

hh_cache 最后面就是 2 层报文头，它会一次性的拷贝到 sk_buff 中，这已经在 ip_finish_output2 见识到了。图中通过虚线告诉各位：这个报文头的内容是从什么地方复制过来的。h_source 是从 net_device 中的 dev_addr 中复制来的，h_dest 是从 neighbour 中的 ha 复制过来，h_proto 是从 hh_type 复制过来的。

下面给出我个人认为最难理解的一段代码，就是 neigh_resolve_output 函数。要注意的是，在报文第一次发出去的时候，它走的不是第 10 行，因为对应邻居的 hh_cache 还没有被创建，只能

直接走到此函数的第 16 行。

```
    在这个函数开头有这么一句注释：/* Slow and careful. */
    那么这个函数有什么要小心的地方呢？读者们会提出疑问：不就几行代码吗，也没有什么奇怪的语法和复
    杂的循环，小心什么呢？
1.  int neigh_resolve_output(struct sk_buff *skb)
2.  {
3.      struct dst_entry *dst = skb->dst;
4.      struct neighbour *neigh;
5.      int rc = 0;
        注意，请看下面对此判断的评解
6.      if (!neigh_event_send(neigh, skb)) {
7.          int err;
8.          struct net_device *dev = neigh->dev;
9.          if (dev->hard_header_cache && !dst->hh) {
10.             if (!dst->hh)
11.                 neigh_hh_init(neigh, dst, dst->ops->protocol);
            不要被这个 err 迷惑了，其实在正常情况下它是 ETH_HLEN，即 14
12.             err = dev->hard_header(skb, dev, ntohs(skb->protocol),
13.                         neigh->ha, NULL, skb->len);
14.         }
            err 肯定大于 0，下面 queue_xmit 才是真正到达了驱动程序那一级代码
15.         if (err >= 0)
16.             rc = neigh->ops->queue_xmit(skb);
17.     }
18. out:
19.     return rc;
20. }
```

代码段 4-27neigh_resolve_output 函数

很显然，我们会进入 neigh_event_send 这个函数，但是要搞清楚返回结果。一定有读者说是 neigh_resolve_output 调用 neigh_event_send 发送了 ARP 报文，那我们看看是否真的如此。先给出 neigh_event_send 的代码：

```
1.  static inline int neigh_event_send(struct neighbour *neigh, struct sk_buff *skb)
2.  {
3.      neigh->used = jiffies;
        ping 127.0.0.1 时，其 nud_state 是 NUD_NOARP，即 0x40，所以这个判断不会进入，只简单返
        回 0。
4.      if (!(neigh->nud_state&(NUD_CONNECTED|NUD_DELAY|NUD_PROBE)))
5.          return __neigh_event_send(neigh, skb);
6.      return 0;
7.  }
```

代码段 4-28 neigh_event_send 函数

然后再进入__neigh_event_send，研究半天，发现这个函数可不象它的名字那样发送一个什么事件，而只是对 neigh→nud_state 进行修改，以期 neigh→timer 能根据这个状态值有进一步操作。

```
1.  int __neigh_event_send(struct neighbour *neigh, struct sk_buff *skb)
2.  {
3.      int rc;
4.      unsigned long now;
5.
6.      rc = 0;
7.      ......
8.
9.      now = jiffies;
10.
11.     if (!(neigh->nud_state & (NUD_STALE | NUD_INCOMPLETE))) {
            arp_tbl 中定义下面这两个成员变量都为 3，所以和为 6，进入条件满足
12.         if (neigh->parms->mcast_probes + neigh->parms->app_probes) {
             把 probes 设置为缺省的 ucast_probes（3），将来在 arp_slocit 函数中使用
13.             atomic_set(&neigh->probes, neigh->parms->ucast_probes);
```

```
14.             neigh->nud_state    = NUD_INCOMPLETE;
15.             neigh->updated = jiffies;
16.
17.             neigh_add_timer(neigh, now + 1);
18.         }
            先不考虑其他情况
19.         ......
20.     } else if (neigh->nud_state & NUD_STALE) {
21.         NEIGH_PRINTK2("neigh %p is delayed.\n", neigh);
22.
23.         neigh->nud_state = NUD_DELAY;
24.         neigh->updated = jiffies;
25.         neigh_add_timer(neigh, jiffies + neigh->parms->delay_probe_time);
26.     }
        由于第 14 行对 nud_state 设置为 INCOMPLETE，所以进入下面的代码段
27.     if (neigh->nud_state == NUD_INCOMPLETE) {
28.         if (skb) {
                queue_len 在 arp_tbl 中设置为 3，如果该 neigh 中待发送的报文超过 3，那么就减少它
29.             if (skb_queue_len(&neigh->arp_queue) >=
30.                 neigh->parms->queue_len) {
31.                 ......
32.                 kfree_skb(buff);
33.             }
      将此报文挂到 neigh 的 arp 报文队列中，必须要注意的是，要发送的 arp 报文并不是这个，而是 arp 模
      块从此报文中抽取相关信息，然后据此再创建一个 arp 报文，我们会在 arp 一节中看到
34.             __skb_queue_tail(&neigh->arp_queue, skb);
35.         }
            返回值为 1，导致 neigh_resolve_output 不会执行第 7 行以下的代码
36.         rc = 1;
37.     }
38. out_unlock_bh:
39.     return rc;
40. }
```

代码段 4-29 __neigh_event_send 函数

读者们注意啦，如果目的地址是 loopback 地址或者本地地址，而本地地址没有 UP 的时候，此函数直接返回 0 到 neigh_resolve_output 函数，进入此函数的第 8 行，不管是 loopback 接口还是普通以太网接口，dev->hard_header_cache 都指向 eth_header_cache，所以都会进入下面这个函数：

```
1.  static void neigh_hh_init(struct neighbour *n, struct dst_entry *dst,
2.            u16 protocol)
3.  {
4.      struct hh_cache *hh;
5.      struct net_device *dev = dst->dev;
6.
7.      for (hh = n->hh; hh; hh = hh->hh_next)
8.          if (hh->hh_type == protocol)
9.              break;
10.
11.     if (!hh && (hh = kzalloc(sizeof(*hh), GFP_ATOMIC)) != NULL) {
12.         rwlock_init(&hh->hh_lock);
13.         hh->hh_type = protocol;
14.         atomic_set(&hh->hh_refcnt, 0);
15.         hh->hh_next = NULL;
        调用 eth_header_cache 函数，几乎所有必要的信息都放在 n 中，只要抽取这些信息放入 hh 中就行
        了
16.         if (dev->hard_header_cache(n, hh)) {
17.             kfree(hh);
18.             hh = NULL;
19.         } else {
20.             atomic_inc(&hh->hh_refcnt);
21.             hh->hh_next = n->hh;
22.             n->hh       = hh;
            对于 ping 127.0.0.1，其 nud_state 是 NUD_NOARP，所以 hh_output 是
            n→ops→hh_output 函数，即 arp_hh_ops 结构中的 dev_queue_xmit 函数
```

```
23.             if (n->nud_state & NUD_CONNECTED)
24.                 hh->hh_output = n->ops->hh_output;
25.             else
26.              hh->hh_output = n->ops->output;
27.         }
28.     }
29.     if (hh) {
     这里对 dst→hh 赋值，它对 ip_finish_output2 中第 19 行产生作用，因为以后的报文发送，将直
     接到达驱动层，而不用 neighbour 系统的帮助了
30.         dst->hh = hh;
31.     }
32. }
```

代码段 4-30 neigh_hh_init 函数

上图中虚线基本上可以描述此函数的功能：h_source 来自于设备的 dev_addr，h_dest 来自于邻居的 ha 地址，而 type 目前就是 ETH_IP，来自于 ipv4_dst_ops。然后代表 3 层信息的 dst 和代表 2 层的 neighbour 都指向了 hh_cache，说明它就是维系 3 层和 2 层的关键！

接着 neigh_resolve_output 给报文填上 2 层报文头，发送到设备层去了。

在此应该有读者开始想到这里面有一个很关键的问题：邻居的 ha 地址是如何得到的？报文都已经发送出去了，怎么回事？这里面隐藏着一个巨大的陷阱。读者必须看完 ARP 那一节之后才能跳出这么一个陷阱。

Ping 环回地址的过程已经在代码中给出，现考虑 ping 一个非环回地址的情况。可以总结如下：

1. 当第一次进入这个函数时，由于我们曾经在创建这个 neighbour{}的时候，设置其状态为 NUD_NONE，所以，可以进入__neigh_event_send
2. 当 neigh→nud_state 被设置为 NUD_INCOMPLETE 后，该 neigh 的定时器函数将要采取一些操作了，这个 timer 是在 neigh_alloc 中制定的。代码列下：

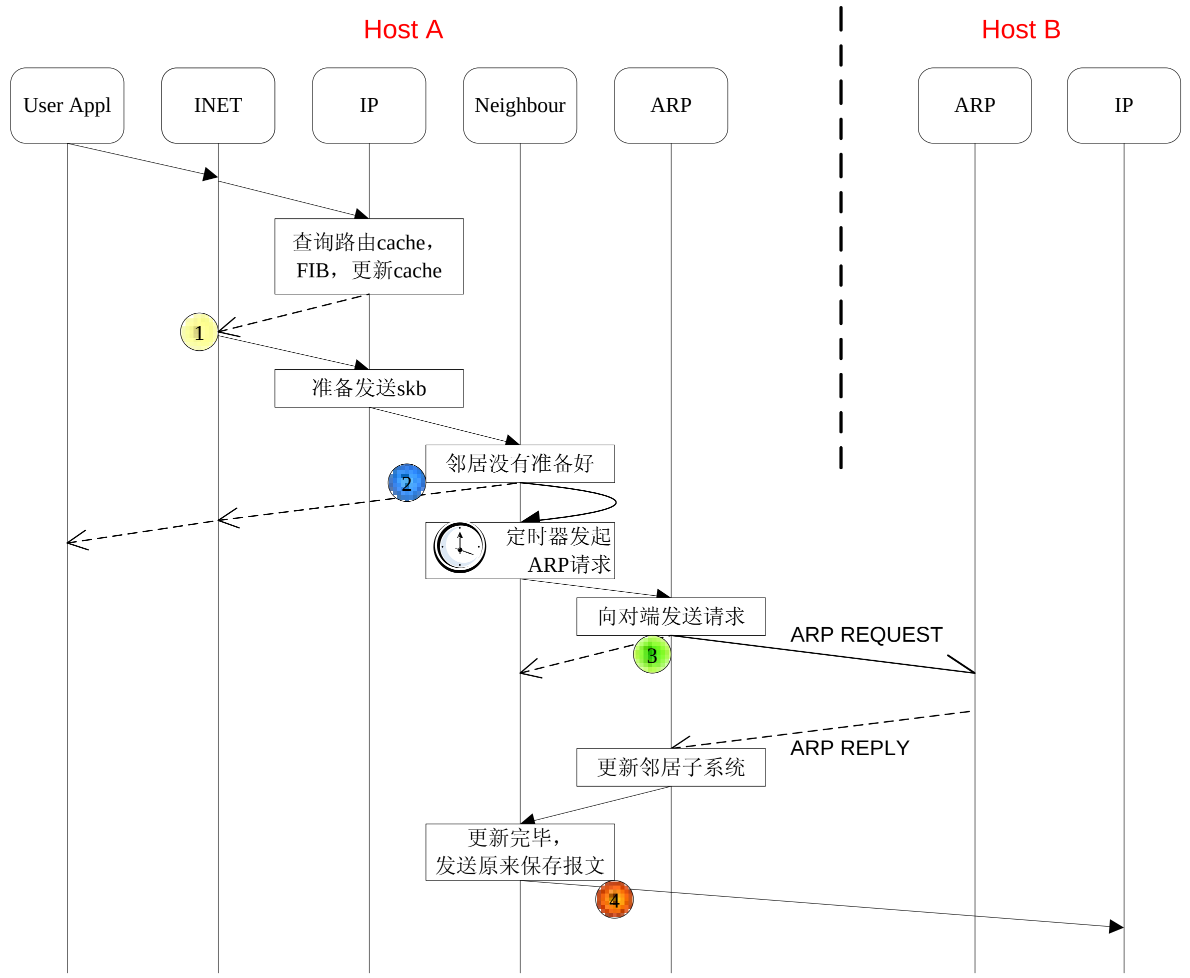
```
1.  /* 此函数只有当邻居表项定时器超时的时候才会被调用
2.  */
3.  static void neigh_timer_handler(unsigned long arg)
4.  {
5.      unsigned long now, next;
6.      struct neighbour *neigh = (struct neighbour *)arg;
7.      unsigned state;
8.      int notify = 0;
9.
    记住我们已经在__neigh_event_send 中将其设置为 NUD_INCOMPLETE
10.     state = neigh->nud_state;
11.     now = jiffies;
12.     next = now + HZ;
    不会进入下面的判断，所以请直接看到第 75 行
13.     if (!(state & NUD_IN_TIMER)) {
14.         goto out;
15.     }
16.
17.     if (state & NUD_REACHABLE) {
18.         if (time_before_eq(now,
19.                 neigh->confirmed + neigh->parms->reachable_time)) {
20.             NEIGH_PRINTK2("neigh %p is still alive.\n", neigh);
21.             next = neigh->confirmed + neigh->parms->reachable_time;
22.         } else if (time_before_eq(now,
23.                    neigh->used + neigh->parms->delay_probe_time)) {
24.             NEIGH_PRINTK2("neigh %p is delayed.\n", neigh);
25.             neigh->nud_state = NUD_DELAY;
26.             neigh->updated = jiffies;
27.             neigh_suspect(neigh);
28.             next = now + neigh->parms->delay_probe_time;
29.         } else {
30.             NEIGH_PRINTK2("neigh %p is suspected.\n", neigh);
```

```
31.             neigh->nud_state = NUD_STALE;
32.             neigh->updated = jiffies;
33.             neigh_suspect(neigh);
34.             notify = 1;
35.         }
36.     } else if (state & NUD_DELAY) {
37.         if (time_before_eq(now,
38.                 neigh->confirmed + neigh->parms->delay_probe_time)) {
39.             NEIGH_PRINTK2("neigh %p is now reachable.\n", neigh);
40.             neigh->nud_state = NUD_REACHABLE;
41.             neigh->updated = jiffies;
42.             neigh_connect(neigh);
43.             notify = 1;
44.             next = neigh->confirmed + neigh->parms->reachable_time;
45.         } else {
46.             NEIGH_PRINTK2("neigh %p is probed.\n", neigh);
47.             neigh->nud_state = NUD_PROBE;
48.             neigh->updated = jiffies;
49.             atomic_set(&neigh->probes, 0);
50.             next = now + neigh->parms->retrans_time;
51.         }
52.     } else {
53.         /* NUD_PROBE|NUD_INCOMPLETE */
54.         next = now + neigh->parms->retrans_time;
55.     }
56.
57.     if ((neigh->nud_state & (NUD_INCOMPLETE | NUD_PROBE)) &&
58.         atomic_read(&neigh->probes) >= neigh_max_probes(neigh)) {
    发送次数超过了neigh最大的刺探数，所以要把队列清掉
59.         struct sk_buff *skb;
60.
61.         neigh->nud_state = NUD_FAILED;
62.         ......
63.         skb_queue_purge(&neigh->arp_queue);
64.     }
65.
66.     if (neigh->nud_state & NUD_IN_TIMER) {
67.         if (time_before(next, jiffies + HZ/2))
68.             next = jiffies + HZ/2;
69.         !mod_timer(&neigh->timer, next);
70.     }
71.     if (neigh->nud_state & (NUD_INCOMPLETE | NUD_PROBE)) {
72.         struct sk_buff *skb = skb_peek(&neigh->arp_queue);
73.         /* keep skb alive even if arp_queue overflows */
74.         if (skb)
75.             skb_get(skb);
76.
   调用arp_solicit，如不记得怎么回事，请参考前面的arp_constructor函数
77.         neigh->ops->solicit(neigh, skb);
   把probes的次数加一，这个变量比较重要，我们会在arp_solicit中引用到
78.         atomic_inc(&neigh->probes);
   难道已经发送完arp报文后，就把队列中的skb删除？不是，实际上只是将报文的引用计数减1，请联系
   第78行的代码（它是对其引用计数加1）
79.         if (skb)
80.             kfree_skb(skb);
81.     } else {
82. out:
83.     }
    奇怪的是，目前Linux系统中没有模块注册对NEIGH_UPDATE事件感兴趣，所以下面的代码没有意义
84.     if (notify)
85.         call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, neigh);
   如果我们在应用层有arp的daemon，就可以如下操作：
86. #ifdef CONFIG_ARPD
87.     if (notify && neigh->parms->app_probes)
88.         neigh_app_notify(neigh);
89. #endif
90.     neigh_release(neigh);
91. }
```

Solicit的中文意思是“恳求”

代码段 4-31 neigh_timer_handler 函数

读者一定搞晕了，那么我们现在就先给出一副图，让大家可以对报文发送的流程有一个总体的概念。注意这里有 2 台主机。



图表 4-10邻居子系统初次发送过程的序列图

正如你所见，系统调用 send 在 ② 处就返回了——用户报文还没有发送出去！而系统内部的定时器开始工作，它发送 ARP 报文给对端主机，当地址解析完毕，然后才把真正的报文发送出去。

4.7 ARP 的作用

I P 地址是运行 T C P / I P 协议机器的通用标识，但是 I P 地址自身不能使报文到达其目的地。网络系统自身及其行为对网络操作系统和硬件类型而言是特殊的。当用户把一块数据发送到另一台机器时，经常通过 I P 地址来完成。虽然 T C P / I P 设计成围绕着 I P 地址工作，但是实际的网络软件和硬件却不是这样。相反，网络使用编码至网络硬件中的地址来识别每一台机器。从 I P 地址得到物理地址，不是 T C P / I P 协议的标准部分，所以开发了许多特殊的协议来完成这一部分任务。

4.7.1 ARP 的机制

ARP（地址解析）协议就是这样一种解析协议，本来主机是完全不知道这个 IP 对应的是哪个主机的哪个接口，当主机要发送一个 IP 包的时候，会首先查一下自己的 ARP 高速缓存（就是一个 IP-MAC 地址对应表缓存，Linux 是通过 rtable+dst_entry+hh_cache 完成的），如果查询的 IP－MAC 值对不存在，那么主机就向网络发送一个 ARP 协议广播包，这个广播包里面就有待查询的 IP 地址，而直接收到这份广播的包的所有主机都会查询自己的 IP 地址，如果收到广播包的某一个主机发现自己符合条件，那么就准备好一个包含自己的 MAC 地址的 ARP 包传送给发送 ARP 广播的主机，而广播主机拿到 ARP 包后会更新自己的 ARP 缓存（就是存放 IP-MAC 对应表的地方）。发送广播的主机就会用新的 ARP 缓存数据准备好数据链路层的的数据包发送工作。

ARP 协议和 RARP 协议

A R P 高效运行的关键是由于每个主机上都有一个 A R P 高速缓存。这个高速缓存存放了最近 I n t e r n e t 地址到硬件地址之间的映射记录。高速缓存中每一项的生存时间一般为 2 0 分钟，起始时间从被创建时开始算起。

A R P 协议有一个缺陷：假如一个设备不知道它自己的 I P 地址，就没有办法产生 A R P 请求和 A R P 应答。一个简单的解决办法是使用反向地址解析协议(RARP)，RARP 以 ARP 相反的方式工作。R A R P 发出要反向解析的物理地址并希望返回其 I P 地址，应答包括由能够提供信息的 R A R P 服务器发出的 I P 地址。虽然发送方发出的是广播信息， RARP 规定只有 RARP 服务器能产生应答。许多网络指定多个 RARP 服务器，这样做既是为了平衡负载也是为了作为出现问题时的备份。

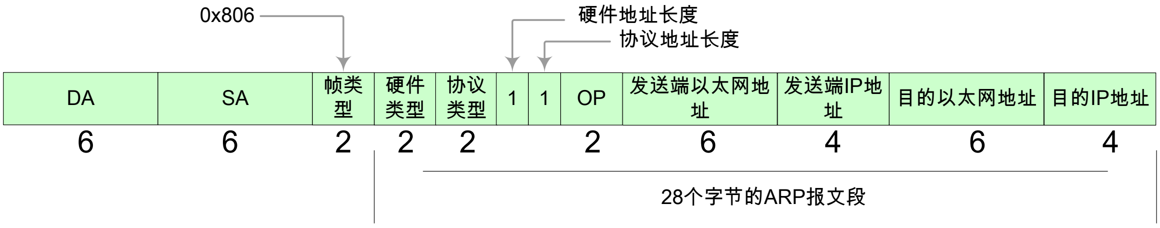
RARP 分组的格式与 ARP 分组基本一致。它们之间主要的差别是 R A R P 请求或应答的帧类型代码为 0 x 8 0 3 5，而且 RARP 请求的操作代码为 3，应答操作代码为 4。

对应于 ARP，RARP 请求以广播方式传送，而 RARP 应答一般是单播(unicast)传送的。

虽然 R A R P 在概念上很简单，但是一个 RARP 服务器的设计与系统相关而且比较复杂。相反，提供一个 A R P 服务器很简单，通常是 T C P / I P 在内核中实现的一部分。由于内核知道 I P 地址和硬件地址，因此当它收到一个询问 I P 地址的 A R P 请求时，只需用相应的硬件地址来提供应答就可以了。

4.7.2 ARP 报文格式

在以太网上解析 I P 地址时， A R P 请求和应答分组的格式如图 4 - 3 所示（ A R P 可以用于其他类型的网络，可以解析 I P 地址以外的地址。紧跟着帧类型字段的前四个字段指定了最后四个字段的类型和长度）。



图表 4-11 ARP 报文格式

以太网报头中的前两个字段是以太网的源地址和目的地址。目的地址为全 1 的特殊地址是广播地址。电缆上的所有以太网接口都要接收广播的数据帧。两个字节长的以太网帧类型表示后面

数据的类型。对于 A R P 请求或应答来说，该字段的值为 0 x 0 8 0 6。

形容词 h a r d w a r e (硬件)和 p r o t o c o l (协议)用来描述 A R P 分组中的各个字段。例如，一个 A R P 请求分组询问协议地址（这里是 I P 地址）对应的硬件地址（这里是以太网地址）。

硬件类型字段表示硬件地址的类型。它的值为 1 即表示以太网地址。协议类型字段表示要映射的协议地址类型。它的值为 0 x 0 8 0 0 即表示 I P 地址。它的值与包含 I P 数据报的以太网数据帧中的类型字段的值相同，这是有意设计的。

接下来的两个 1 字节的字段，硬件地址长度和协议地址长度分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 I P 地址的 A R P 请求或应答来说，它们的值分别为 6 和 4。

操作字段指出四种操作类型，它们是 A R P 请求（值为 1）、A R P 应答（值为 2）、R A R P 请求（值为 3）和 R A R P 应答（值为 4）（我们在第 5 章讨论 R A R P)。这个字段必需的，因为 A R P 请求和 A R P 应答的帧类型字段值是相同的。

接下来的四个字段是发送端的硬件地址（在本例中是以太网地址）、发送端的协议地址（I P 地址）、目的端的硬件地址和目的端的协议地址。注意，这里有一些重复信息：在以太网的数据帧报头中和 A R P 请求数据帧中都有发送端的硬件地址。

对于一个 A R P 请求来说，除目的端硬件地址外的所有其他的字段都有填充值。当系统收到一份目的端为本机的 A R P 请求报文后，它就把硬件地址填进去，然后用两个目的端地址分别替换两个发送端地址，并把操作字段置为 2，最后把它发送回去。

从 RFC 的定义上看，我们可以归纳出一个系统如果要实现 ARP，必须提供一个 arp table 结构以完成 ARP 的协议。那么此结构应该有如下字段:

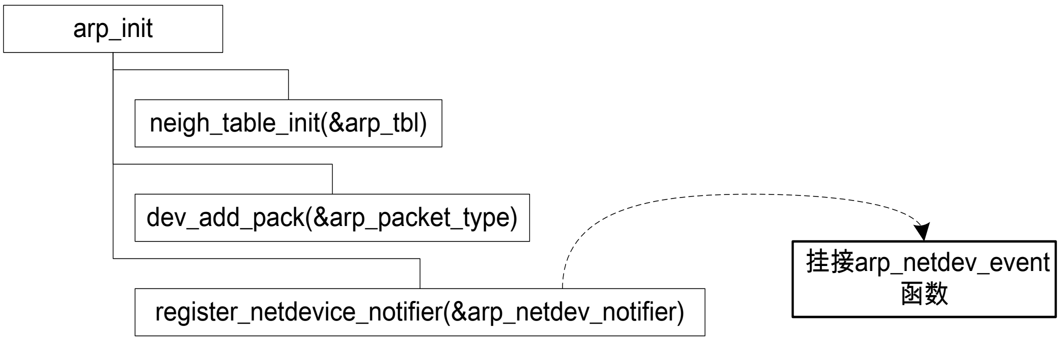
- ✓ last used 本 ARP 项最近一次使用的时间
- ✓ last updated 本 ARP 项最近一次更新的时间
- ✓ flags 描述本项的状态，如是否完成等
- ✓ IP address 本项描述的 IP 地址
- ✓ hardware address 要解析的硬件地址
- ✓ hardware header 指向缓存硬件头的指针
- ✓ timer 是个 timer_list 项，用于 ARP 请求没有响应时的超时
- ✓ retries ARP 请求重试的次数
- ✓ sk_buff queue 等待 IP 地址解析的 sk_buff 项列表

那么 Linux 是如何实现的呢？

4.7.3 Linux ARP 协议的实现

4.7.3.1. 协议初始化

我们之前在系统初始化的时候已经看到 arp 的初始化了,现在详细讨论一下它的内部实现，还是从 arp_init 开始。



图表 4-12　arp_init 函数调用树

先介绍系统中的一个重要的数据结构：neigh_table。它是用来包含和本机相连接的所有的邻居的数据结构。这里表示的连接表示的是物理连接，如果没有任何机器和这台机器相连接的花，那么可以不初始化这个数据结构；如果是一个星型网络的中心，那么和它相连的所有机器都会表示为这种形式的数据存放在 neigh_table 结构的链表中。

arp_tbl 是一个类型为 struct neigh_table 的全局变量，它是一个 ARP 的缓存表，也称为邻居表。协议栈通过 ARP 协议获取到的网络上邻居主机的 IP 地址与 MAC 地址的对应关系都会保存在这个表中，以备下次与邻居通讯时使用，同时，ARP 模块自身也会提供一套相应的机制来更新和维护这个邻居表。下面逐个分析 arp_tbl 中的重要成员数据与函数。

对于 TCP/IP 协议来说，应该是以 ARP 方式判断邻居，即 ARP 数据包到达的下一台机器就是本机的邻居，应该加入到 ARP 指定的一个 neigh_table{ }结构中，也就是 arp_tbl 变量。注：2.6 中无 gc_task 变量了。

```
1.  /*
2.   *  neighbour table manipulation
3.   */
4.
5.  struct neigh_table
6.  {
7.      struct neigh_table *next;
        对于 arp_tbl 而言,只能是 AF_INET
8.      int         family;
        entry_size 是一个入口的大小，也就是 arp_tbl 中一个邻居的大小，邻居用 struct neighbour 结构体表示，该结构体的最后一个成员是 u8 primary_key[0]，用于存放 IP 地址，作为这个邻居的哈希主键。所以 entry_size 的大小就是 sizeof(struct neighbour) + 4，因为是用 IP 地址作主键，所以 key_len 就是 4。kmem_cachep 是一个后备高速缓存，创建一个邻居需要的内存从这个后备高速缓存中去取。
9.      int         entry_size;
10.     int         key_len;
11.     kmem_cache_t    *kmem_cachep;
        hash_buckets 是一个哈希数组，里面存放了 arp_tbl 当前维护的所有的邻居，hash_mask 是哈希数组大小的掩码，其初始值为 1，所以 hash_buckets 的初始大小为 2(0 到 hash_mask 的空间范围)。entries 是整个 arp_tbl 中邻居的数量，当 entries 大于 hash_mask+1 的时候，hash_buckets 增长为原来的两部。成员 hash 是一个哈希函数指针，用于计算哈希值。
12.     __u32       (*hash)(const void *pkey, const struct net_device *);
13.     struct neighbour    **hash_buckets;
14.     unsigned int       hash_mask;
15.     atomic_t        entries;
        这是用于代理 ARP 的邻居哈希表，固定有 PNEIGH_HASHMASK 项即 0xF 个，其它与 hash_buckets 相同。
16.     struct pneigh_entry**phash_buckets;
        这是一个邻居的初始化函数指针，每次创建出一个邻居后，需要马上调用这个函数对新创建的邻居进行一些初始化操作。邻居创建完，已经被赋于一个 IP 地址(邻居结构体的 primary_key 成员)，该函数首先
```

根据这个 IP 地址来确定其地址类型，然后为邻居选择相应的操作函数集(初始化邻居结构体的一些成员，在讲到邻居结构体内容时再进行分析)。

```
17.     int         (*constructor)(struct neighbour *);
```

这是代理 ARP 的邻居的构建和析构函数指针，在 IPv4 模块中，未提供这两个函数，所以它们的指针值为空。

```
18.     int         (*pconstructor)(struct pneigh_entry *);
19.     void        (*pdestructor)(struct pneigh_entry *);
20.     void        (*proxy_redo)(struct sk_buff *skb);
```

id 作为这个邻居表的一个名称，是一个字符串信息，内核协议栈的 arp_tbl 的 id 是 arp_cache。

```
21.     char        *id;
```

这是一个结构体 neigh_parms{}的链表，系统中每个网络设备接口对应链表中一个节点，表示该设备接口上的邻居的一些传输参数。同时，链表中还有一个缺省的项。

```
22.     struct neigh_parms  parms;
```

gc_thresh3 是 arp_tbl 中允许拥有的邻居数量的上限，一旦超过这个上限，并且表中没有可以清理掉的垃圾邻居，那么就无法创建新的邻居，这个值缺省被置为 1024。gc_thresh2 是第二个阀值，如果表中的邻居数量超过这个阀值，并且在需要创建新的邻居时，发现已经超过 5 秒时间表没有被刷新过，则必须立即刷新 arp_tbl 表，进行强制垃圾回收，这个值缺省被置为 512。gc_thresh1 的用途暂时还没有发现，它缺省被置为 128。gc_interval 应该是常规的垃圾回收间隔时间，被缺省置为 30 秒，但目前在源代码中似乎没有看到它的应用。强制垃圾收集的工作即是把引用计数为 1，且状态中没有 NUD_PERMANENT 的邻居全部从 arp_tbl 表中删除。

```
23.     int         gc_interval;
24.     int         gc_thresh1;
25.     int         gc_thresh2;
26.     int         gc_thresh3;
27.     unsigned long       last_flush;
```

这是一个常规垃圾回收的定时器，其定时处理函数是 neigh_periodic_timer。该定时器超时后，处理函数处理 hash_buckets 表中的一项，下次超时后，再处理下一项，这里的垃圾回收比强制垃圾回收条件要宽松得多，如果邻居的状态为 NUD_PERMANENT 或 NUD_IN_TIMER(该邻居正在解析中)，则不能回收。当邻居的引用计数为 1 时，并且邻居状态为 NUD_FAILED(解析失败)或者该邻居距最近一次被使用时间已超过参数表中 gc_staletime 的值(缺省为 60 秒),则可以作为垃圾回收。回收完毕后，要设置下一次进行回收的时间(gc_timer 的超时时间)，下次回收时间为参数表中 base_reachable_time 的值(缺省设为 30 秒)的一半，再除以 hash_buckets 哈希表中的项数。也就是，基本上 15 秒左右会把整个 arp_tbl 缓存表进行一次垃圾回收。

```
28.     struct timer_list   gc_timer;
```

proxy_timer 是一个关于代理 ARP 的定时器，proxy_queue 是一个待处理的代理 ARP 数据包的队列，每次定时器超时，处理函数 neigh_proxy_process 依次检查队列中每一个代理 ARP 数据包(struct sk_buff)，对于超时，且满足相关条件的，调用 proxy_redo 进行处理。有关代理 ARP，这里暂时略过。

```
29.     struct timer_list   proxy_timer;
30.     struct sk_buff_head proxy_queue;
31.     rwlock_t        lock;
```

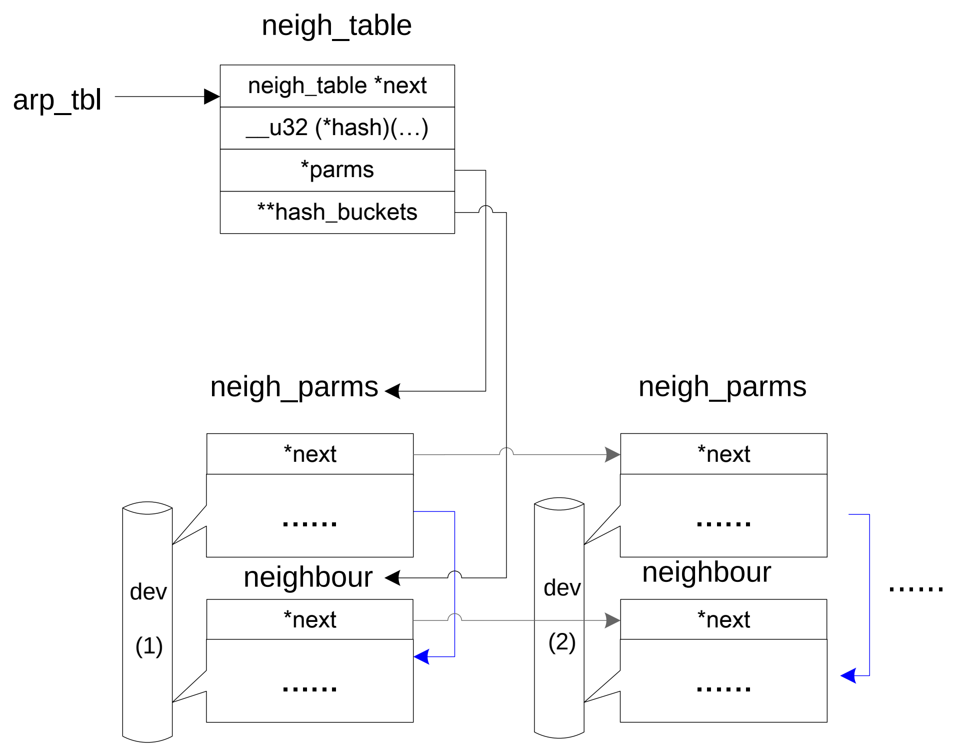
这两个成员其实没有联系，hash_rand 是用于邻居哈希表 hash_buckets 的一个随机数，last_rand 用于记录一个时间，即上次为 parms 链表中每个节点生成 reachable_time 的时间，reachable_time 是需要被定时刷新的。

```
32.     unsigned long       last_rand;
33.     __u32           hash_rnd;
```

记录 arp_tbl 被操作次数的一些统计数据。

```
34.     struct neigh_statistics*stats;
35.     unsigned int        hash_chain_gc;
36. };
```

代码段 4-32　neigh_table 结构



图表 4-13neigh_table 各成员的关系图

其中有一个结构叫 neigh_parms，我们之前在配置接口地址的时候（inetdev_init 函数中）曾经见到它被创建，也就是说，此结构是跟着接口走。有多少个接口，就有多少个 neigh_parms，它存放对邻居表进行配置的信息。由于每一个接口的邻居系统可以单独配置，这样就非常灵活了。比如 A 接口要求 1 分钟刷新一次邻居表，而 B 接口也许要求 3 分钟刷新。这就是 neigh_params 的用处：

```
1.  struct neigh_parms
2.  {
3.      struct net_device *dev;
4.      struct neigh_parms *next;
5.      int (*neigh_setup)(struct neighbour *);
6.      void    (*neigh_destructor)(struct neighbour *);
7.      struct neigh_table *tbl;
8.  
9.      void    *sysctl_table;
10. 
11.     int dead;
12.     atomic_t refcnt;
13.     struct rcu_head rcu_head;
14. 
15.     int base_reachable_time;
16.     int retrans_time;
17.     int gc_staletime;
18.     int reachable_time;
19.     int delay_probe_time;
20. 
21.     int queue_len;
22.     int ucast_probes;
23.     int app_probes;
24.     int mcast_probes;
25.     int anycast_delay;
26.     int proxy_delay;
27.     int proxy_qlen;
28.     int locktime;
```

```
29. }
```

代码段 4-33 neigh_parms 结构

在 arp_init 中调用了 neigh_table_init，参数为 arp_tbl，该结构定义如下：

```
1.  struct neigh_table arp_tbl = {
2.      .family =   AF_INET,
3.      .entry_size =   sizeof(struct neighbour) + 4,
4.      .key_len =  4,
5.      .hash =     arp_hash,
6.      .constructor = arp_constructor,
7.      .proxy_redo =   parp_redo,
8.      .id =       "arp_cache",
     在这里就初始化了 neigh_parms 结构
9.      .parms = {
     tbl 就指向 arp_tbl 自己，在绝大部分情况，下面
     这个结构的值会全部复制到接口自己的 neigh_parar
     中，如右边的函数所做。
10.         .tbl =          &arp_tbl,
11.         .base_reachable_time = 30 * HZ,
12.         .retrans_time = 1 * HZ,
13.         .gc_staletime = 60 * HZ,
14.         .reachable_time =  30 * HZ,
15.         .delay_probe_time = 5 * HZ,
16.         .queue_len =       3,
17.         .ucast_probes = 3,
18.         .mcast_probes = 3,
19.         .anycast_delay = 1 * HZ,
20.         .proxy_delay = (8 * HZ) / 10,
21.         .proxy_qlen =  64,
22.         .locktime = 1 * HZ,
23.     },
24.     .gc_interval = 30 * HZ,
25.     .gc_thresh1 =  128,
26.     .gc_thresh2 =  512,
27.     .gc_thresh3 =  1024,
28. };
```

```
struct  neigh_parms  *neigh_parms_alloc(struct
net_device *dev,
                      struct neigh_table *tbl)
{
    struct neigh_parms *p = kmalloc(......);

    if (p) {
        memcpy(p, &tbl->parms, sizeof(*p));
        p->tbl       = tbl;

        if (dev) {
            p->dev = dev;
        }
        p->sysctl_table = NULL;
        p->next     = tbl->parms.next;
        tbl->parms.next = p;
    }
    return p;
}
```

代码段 4-34 neigh_table 结构

ARP 表包括了指向 arp_table 链的指针（ arp_table 向量）。缓存这些表项可以加速对它们的访问，每个表项用 IP 地址的最后两个字节来生成索引，然后就可以查找表链以找到正确的表项。Linux 也以 hh_cache 结构的形式来缓存 arp_table 项的预建的硬件头。

请求一个 IP 地址解析并且没有相应的 arp_table 项时，ARP 必须发送一个 ARP 请求。它在表和 sk_buff 队列中 生成一个新的 arp_table 项， sk_buff 包含了需要进行地址解析的网络包。发送 ARP 请求时运行 ARP 定时器。如果没有响应，ARP 将重试几次，如果仍然没有响应，ARP 将删除该 arp_table 项。同时会通知队列中等待 IP 地址解析的 sk_buff 结构，传送它们的上层协议将处理这一失败。UDP 不关心丢包，而 TCP 则会建立 TCP 连接进行重传。如果 IP 地址的所有者返回了它的硬件地址，则 arp_table 项被标记为完成，队列中的 sk_buff 将被删除，传输动作继续。硬件地址被写到每个 sk_buff 的硬件头中。

ARP 协议层必须响应 ARP 请求。它注册它的协议类型(ETH_P_ARP)，生成一个 packet_type 结构。这表示它将检查网络设备收到的所有 ARP 包。与 ARP 应答一样，这包括 ARP 请求。用保存在接收设备的 device 结构中的硬件地址来生成 ARP 应答。

网络拓扑结构会随时间改变，IP 地址会被重新分配不同的硬件地址。例如，一些拨号服务为每一次新建的连接分配一个 IP 地址。为了使 ARP 表包含这些数据项，ARP 运行一个周期性的定

时器，用来查看所有的 arp_table 项中哪一个。

neigh_table_init 在内核中有多处，但仔细研究大多是和 IP 网没有关系的。所以唯一和 IP 网有关系的初始化是在 arp_init 函数中调用的，函数代码如下：

```
1.  void neigh_table_init(struct neigh_table *tbl)
2.  {
        此处的 tbl 就是 arp_tbl
3.      struct neigh_table *tmp;
4.  
5.      neigh_table_init_no_netlink(tbl);
6.  
        neigh_tables 是一个全局指针，如果系统中只有 IP 协议栈，那么就只有一个 arp_tbl
7.      for (tmp = neigh_tables; tmp; tmp = tmp->next) {
8.          if (tmp->family == tbl->family)
9.              break;
10.     }
11.     tbl->next   = neigh_tables;
12.     neigh_tables    = tbl;
13. }
```

代码段 4-35 neigh_table_init 函数

dev_add_pack 是一个非常重要的函数，在 arp_init 中传入的参数为 arp_packet_type；ip_init 中也调用了它，给它传入的参数是 ip_packet_type，这个变量的类型是 packet_type：

当一个邻居刚刚被创建，其状态是 NUD_NONE，此时，邻居被创建出来，并根据其 IP 地址被加入到 arp_tbl 的哈希表 hash_buckets 中，但它还没有被解析，其成员 ha(邻居的硬件地址)为空。直到向这个邻居发送 IP 数据报，并且发送流程到达网络层的最后一个发送函数 ip_finish_output2 时，调用邻居的 output 成员函数，一般这个函数是 neigh_resolve_output(根据 type 的不同会略有不同)。它会先进行 ARP 解析，然后把 IP 数据报发往正确的邻居。

4.7.3.2. 开始 ARP 状态机

因为当前状态是 NUD_NONE，neigh_resolve_output 首先判断 parms 的成员 mcast_probes 加上 app_probes 的值，这两个参数表示 ARP 解析时尝试的次数，mcast_probes 缺省值置为 3，app_probes 是 0，如果它们的和为零，则不作 ARP 解析尝试，直接将状态置为 NUD_FAILED。当前不为零，所以进行解析，首先置邻居的成员 probes 为 parms 的 ucast_probes，缺省为 3，这样，该邻居等于是已经尝试了 3 次了，另外还只能多 3 次(mcast_probes)，即该邻居的解析最多尝试 3 次，如果 3 次不成功，则失败，结束。然后将状态置为 NUD_INCOMPLETE，表示正在解析中，并把待解析的 socket 缓冲 skb 放入 arp_queue 队列中，然后启动邻居的定时器开始 ARP 解析。

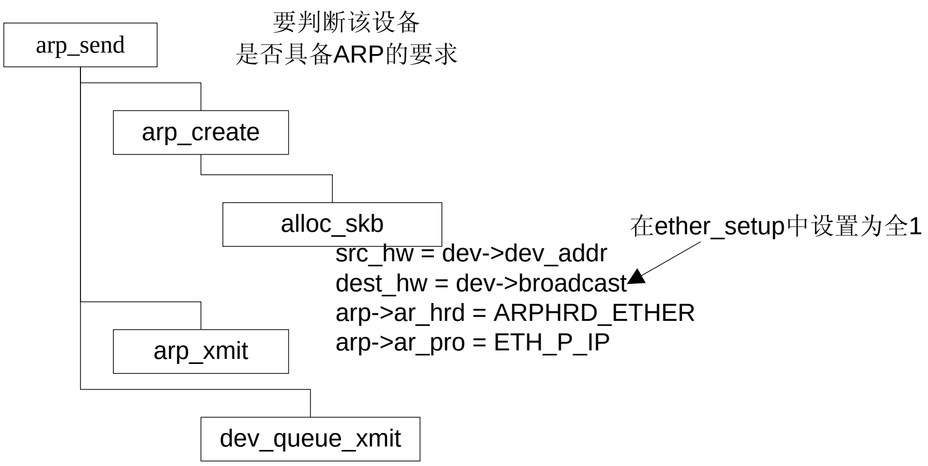
定时器处理函数 neigh_timer_handler 首先确定下次的超时时间(如果这次解析没有成功)为当前时间加上 parms 的成员 retrans_time 的值（缺省设置为 1 秒)。所以，ARP 解析的发包超时时间是 1 秒，连续尝试 3 次。ARP 解析是通过 ops 的成员函数 solicit 去做的，该成员函数指针指向的 arp_solicit 函数，arp_solicit 会实际发送 ARP 请求包。

```
1.  static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)
2.  {
3.      u32 saddr = 0;
4.      u8  *dst_ha = NULL;
5.      struct net_device *dev = neigh->dev;
6.      u32 target = *(u32*)neigh->primary_key;
    根据前面__neigh_event_send 的处理，probes 等于 3
7.      int probes = atomic_read(&neigh->probes);
8.      struct in_device *in_dev = in_dev_get(dev);
9.  
10.     switch (IN_DEV_ARP_ANNOUNCE(in_dev)) {
```

```
11.     default:
12.     case 0:     /* By default announce any local IP */
13.         if (skb && inet_addr_type(skb->nh.iph->saddr) == RTN_LOCAL)
14.             saddr = skb->nh.iph->saddr;
15.         break;
16.     case 1:     /* Restrict announcements of saddr in same subnet */
17.         if (!skb)
18.             break;
19.         saddr = skb->nh.iph->saddr;
20.         if (inet_addr_type(saddr) == RTN_LOCAL) {
21.             /* saddr should be known to target */
22.             if (inet_addr_onlink(in_dev, target, saddr))
23.                 break;
24.         }
25.         saddr = 0;
26.         break;
27.     case 2:     /* Avoid secondary IPs, get a primary/preferred one */
28.         break;
29.     }
30.
31.     if (!saddr)
32.         saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);
        ucast_probes 缺省是 3
33.     if ((probes -= neigh->parms->ucast_probes) < 0) {
34.         if (!(neigh->nud_state&NUD_VALID))
35.             printk(KERN_DEBUG "trying to ucast probe in NUD_INVALID\n");
36.         把目的地址指向邻居的 mac 地址
37.         dst_ha = neigh->ha;
38.     } else if ((probes -= neigh->parms->app_probes) < 0) {
39. #ifdef CONFIG_ARPD
40.         neigh_app_ns(neigh);
41. #endif
42.         return;
43.     }
        发送 ARP 请求报文
44.     arp_send(ARPOP_REQUEST, ETH_P_ARP,
45.             target, dev, saddr, dst_ha, dev->dev_addr, NULL);
46. }
```

代码段 4-36 arp_solicit 函数

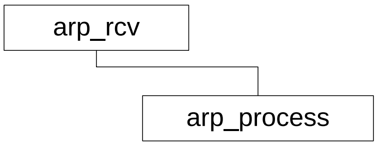
下图中，arp_send 调用 arp_create 函数，它不仅申请 skb，还要把这个 skb 初始化，其值主要根据 dev 来获取，要注意的是 dev→broadcast 是一个全 1 的值，即 0xFFFFFFFF，它是在 ether_setup 函数中赋值。



图表 4-14 arp_send 函数调用树

如图中说述，判断设备是否具备 ARP 的条件只需检查 dev→flags 是否是 NOARP 的，如果是，就直接返回，这种设备就是我们知道的 X25 设备和 ISDN 设备等等。到这里，一个 ARP 报文就发送到设备驱动了。从叙述的顺序来看，我得介绍驱动层的实现了。不过，为了让话题集中在 ARP，我必须避开设备发送的机理，重点在于谈 ARP 参与者之间交互以及 ARP 与邻居子系统交互的机制。同样的道理，我们先不说主机是如何将报文收到协议栈来的，大家记住 ARP 接收报文的第一个函数是 arp_rcv 就可以了。

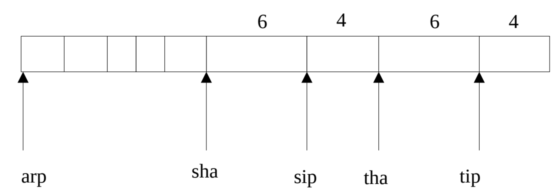
好，发送和接收的底层实现我们就放到后面去说。现在来看看当主机 A 发送报文到主机 B，这时主机 B 用 arp_rcv 函数把报文收了上了，它该做些什么：



图表 4-15 arp_rcv 函数调用树

掐头去尾，该函数调用了 arp_process 函数，我们应该把重点放在这个函数身上。

```
1.  static int arp_process(struct sk_buff *skb)
2.  {
3.      struct net_device *dev = skb->dev;
4.      struct in_device *in_dev = in_dev_get(dev);
5.      struct arphdr *arp;
6.      unsigned char *arp_ptr;
7.      struct rtable *rt;
8.      unsigned char *sha, *tha;
9.      u32 sip, tip;
10.     u16 dev_type = dev->type;
11.     int addr_type;
12.     struct neighbour *n;
13.
14. /* arp_rcv below verifies the ARP header and verifies the device
15.  * is ARP'able.
16.  */
17.
18.     arp = skb->nh.arph;
19.
20.     switch (dev_type) {
21.     default:
        如果不是为 IP 协议或者两个接口类型不匹配，则必须退出
22.         if (arp->ar_pro != htons(ETH_P_IP) ||
23.             htons(dev_type) != arp->ar_hrd)
24.              goto out;
25.         break;
26.     ......
27.     }
28. /* 检查消息类型 */
29.     if (arp->ar_op != htons(ARPOP_REPLY) &&
30.         arp->ar_op != htons(ARPOP_REQUEST))
31.          goto out;
32.
33. /*
34.  *   Extract fields
35.  */
36.     arp_ptr= (unsigned char *)(arp+1);
37.     sha  = arp_ptr;
38.     arp_ptr += dev->addr_len;
39.     memcpy(&sip, arp_ptr, 4);
40.     arp_ptr += 4;
41.     tha  = arp_ptr;
42.     arp_ptr += dev->addr_len;
```



41 行～48 行要完成的工作

```
43.    memcpy(&tip, arp_ptr, 4);
44. /*
45.  *  Check for bad requests for 127.x.x.x and requests for multicast
46.  *  addresses.  If this is one such, delete it.
47.  */
48.    if (LOOPBACK(tip) || MULTICAST(tip))
49.        goto out;
50. ......
51. /*
52.  *  Process entry.  The idea here is we want to send a reply if it is a
53.  *  request for us or if it is a request for someone else that we hold
54.  *  a proxy for.  We want to add an entry to our cache if it is a reply
55.  *  to us or if it is a request for our address.
56.  *  (The assumption for this last is that if someone is requesting our
57.  *  address, they are probably intending to talk to us, so it saves time
58.  *  if we cache their address.  Their address is also probably not in
59.  *  our cache, since ours is not in their cache.)
60.  *
61.  *  Putting this another way, we only care about replies if they are to
62.  *  us, in which case we add them to the cache.  For requests, we care
63.  *  about those for us and those for our proxies.  We reply to both,
64.  *  and in the case of requests for us we add the requester to the arp
65.  *  cache.
66.  */
67.
68. /* Special case: IPv4 duplicate address detection packet (RFC2131) */
69.    if (sip == 0) {
70.        if (arp->ar_op == htons(ARPOP_REQUEST) &&
71.            inet_addr_type(tip) == RTN_LOCAL &&
72.            !arp_ignore(in_dev,dev,sip,tip))
73.                arp_send(ARPOP_REPLY,ETH_P_ARP,
74.                         tip,dev,tip,sha,dev->dev_addr,dev->dev_addr);
75.        goto out;
76.    }
77.
78.    if (arp->ar_op == htons(ARPOP_REQUEST) &&
79.        ip_route_input(skb, tip, sip, 0, dev) == 0) {
```
能够进入这段代码说明此时这是一个 ARP 的接收方，正常情况下它收到了一个 ARP 的请求报文，并且在

```
80.        rt = (struct rtable*)skb->dst;
81.        addr_type = rt->rt_type;
82.
83.        if (addr_type == RTN_LOCAL) {
```
发现该请求报文的目的地址正是本主机

```
84.            n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
85.            if (n) {
86.                int dont_send = 0;
87.
88.                if (!dont_send)
89.                    dont_send |= arp_ignore(in_dev,dev,sip,tip);
90.                if (!dont_send && IN_DEV_ARPFILTER(in_dev))
91.                    dont_send |= arp_filter(sip,tip,dev);
92.                if (!dont_send)
```
发送 ARP 回应给对方主机

```
93.                    arp_send(ARPOP_REPLY,ETH_P_ARP,
94.                             sip,dev,tip,sha,dev->dev_addr,sha);
95.            neigh_release(n);
96.            }
```
发送回应之后，立即退出

```
97.            goto out;
98.        } else if (IN_DEV_FORWARD(in_dev)) {
```
下面几行代码是要求用 proxy ARP 的，这超过了本书的范围,所以省略

```
99.            ......
100.                   goto out;
101.               }
102.           }
103.       }
104.
105.       /* Update our ARP tables */
```
当收到的报文不是 REQUEST 时才会继续走下去，先查询邻居，内部不要要创建 neigh，因为最后一个参

```
       数是 0
106.        n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
107.
108.        if (ipv4_devconf.arp_accept) {
       收到一个不是主动要求的 ARP 回应报文，缺省情况下不会进入本段
109.            /* Unsolicited ARP is not accepted by default.
110.               It is possible, that this option should be enabled for some
111.               devices (strip is candidate)
112.             */
113.            if (n == NULL &&
114.                arp->ar_op == htons(ARPOP_REPLY) &&
115.                inet_addr_type(sip) == RTN_UNICAST)
116.                 n = __neigh_lookup(&arp_tbl, &sip, dev, -1);
117.        }
118.
119.        if (n) {
            如果查到了，就设置对端是 REACHABLE 的，然后更新自己的邻居系统
120.            int state = NUD_REACHABLE;
121.            int override;
122.
123.            /* If several different ARP replies follows back-to-back,
124.               use the FIRST one. It is possible, if several proxy
125.               agents are active. Taking the first reply prevents
126.               arp trashing and chooses the fastest router.
127.             */
128.            override = time_after(jiffies, n->updated + n->parms->locktime);
129.
130.            /* Broadcast replies and request packets
131.               do not assert neighbour reachability.
132.             */
133.            if (arp->ar_op != htons(ARPOP_REPLY) ||
134.                skb->pkt_type != PACKET_HOST)
135.                 state = NUD_STALE;
136.
137.            neigh_update(n, sha, state, override ? NEIGH_UPDATE_F_OVERRIDE : 0);
138.            neigh_release(n);
139.        }
140.
141.    out:
142.    if (in_dev)
143.        in_dev_put(in_dev);
144.    kfree_skb(skb);
145.    return 0;
146. }
```

代码段 4-37 arp_process 函数

我们只关注收到的 ARP 回应包。ARP 回应包中的发送端 IP 地址即为邻居的 IP 地址，我们以这个 IP 地址为主键到 ARP 缓存 arp_tbl 中去寻找这个邻居节点(hash_buckets 成员)，因为在发送 ARP 请求之前，进行邻居绑定的时候，我们已经建立了一个未被解析的邻居，所以这次寻找肯定是成功的，如果没有找到，则直接放弃，不作处理。

找到了这个邻居，我们就要用新的 ARP 回应包的内容刷新这个邻居的内容，但为了防止邻居被过度频繁刷新，引起抖动，只有距邻居上次刷新时间超过 parms->locktime(缺省置为 1 秒)后才允许再次刷新，这是一个可配置的项，可通过修改文件/proc/sys/net/ipv4/neigh/设备名/locktime 进行修改。

最终会调用 neigh_update 去更新邻居子系统。新的邻居状态为 NUD_REACHABLE，该函数中首先更新邻居的成员数据 confirmed 和 updated 为当前时间，然后删除邻居的定时器（定时器正在等待下一次重发 ARP 请求)，把定时器修改到距当前时间 parms->reachable_time(初始设置为 30 秒，实际值在 15-45 秒之间随机变动)。并把新的 MAC 地址复制到邻居的成员 ha 中。

```
1.  /* 通用的更新函数.
2.     -- lladdr 是新的 lladdr，也可以是 NULL,
3.     -- new    新的状态.
4.     -- flags
5.  NEIGH_UPDATE_F_OVERRIDE：如果不相同，允许覆盖给存在的 lladdr,
6.  NEIGH_UPDATE_F_WEAK_OVERRIDE： will suspect existing "connected"
7.              lladdr instead of overriding it
8.              if it is different.
9.              It also allows to retain current state
10.             if lladdr is unchanged.
11. NEIGH_UPDATE_F_ADMIN：意味着此次更新是由管理员触发的
12. NEIGH_UPDATE_F_OVERRIDE_ISROUTER allows to override existing
13.             NTF_ROUTER flag.
14. NEIGH_UPDATE_F_ISROUTERindicates if the neighbour is known as
15.             a router.
16.
17.    调用者必须在入口控制引用计数
18.  */
19.
20. int neigh_update(struct neighbour *neigh, const u8 *lladdr, u8 new,
21.      u32 flags)
22. {
23.     u8 old;
24.     int err;
25. #ifdef CONFIG_ARPD
26.     int notify = 0;
27. #endif
28.     struct net_device *dev;
29.     int update_isrouter = 0;
30.
31.     dev    = neigh->dev;
32.     old    = neigh->nud_state;
33.     err    = -EPERM;
34.
35.     if (!(flags & NEIGH_UPDATE_F_ADMIN) &&
36.     (old & (NUD_NOARP | NUD_PERMANENT)))
37.         goto out;
38.
39.     if (!(new & NUD_VALID)) {
40.         neigh_del_timer(neigh);
41.         if (old & NUD_CONNECTED)
42.             neigh_suspect(neigh);
43.         neigh->nud_state = new;
44.         err = 0;
45. #ifdef CONFIG_ARPD
46.         notify = old & NUD_VALID;
47. #endif
48.         goto out;
49.     }
50.
51. /* Compare new lladdr with cached one */
52.     if (!dev->addr_len) {
53.      /* First case: device needs no address. */
54.         lladdr = neigh->ha;
55.     } else if (lladdr) {
56.      /* The second case: if something is already cached
57.         and a new address is proposed:
58.         - compare new & old
59.         - if they are different, check override flag
60.       */
61.         if ((old & NUD_VALID) &&
62.             !memcmp(lladdr, neigh->ha, dev->addr_len))
63.             lladdr = neigh->ha;
64.     } else {
65.      /* No address is supplied; if we know something,
66.         use it, otherwise discard the request.
67.       */
68.         err = -EINVAL;
```

```
69.         if (!(old & NUD_VALID))
70.             goto out;
71.         lladdr = neigh->ha;
72.     }
73.
74.     if (new & NUD_CONNECTED)
75.         neigh->confirmed = jiffies;
76.     neigh->updated = jiffies;
77.
78. /* If entry was valid and address is not changed,
79.    do not change entry state, if new one is STALE.
80.  */
81.     err = 0;
82.     update_isrouter = flags & NEIGH_UPDATE_F_OVERRIDE_ISROUTER;
83.     if (old & NUD_VALID) {
84.         if (lladdr != neigh->ha && !(flags & NEIGH_UPDATE_F_OVERRIDE)) {
85.              update_isrouter = 0;
86.             if ((flags & NEIGH_UPDATE_F_WEAK_OVERRIDE) &&
87.                 (old & NUD_CONNECTED)) {
88.                 lladdr = neigh->ha;
89.                 new = NUD_STALE;
90.             } else
91.                 goto out;
92.         } else {
93.             if (lladdr == neigh->ha && new == NUD_STALE &&
94.                 ((flags & NEIGH_UPDATE_F_WEAK_OVERRIDE) ||
95.                  (old & NUD_CONNECTED)))
96.                 new = old;
97.         }
98.     }
99.
100.        if (new != old) {
101.            neigh_del_timer(neigh);
102.            if (new & NUD_IN_TIMER) {
103.
104.                neigh_add_timer(neigh, (jiffies +
105.                            ((new & NUD_REACHABLE) ?
106.                             neigh->parms->reachable_time :
107.                             0)));
108.            }
109.            neigh->nud_state = new;
110.        }
111.
112.        if (lladdr != neigh->ha) {
            读者们请注意了，这次内存拷贝就是回答上一节的问题：neigh→ha 是从 lladdr 而来。
113.            memcpy(&neigh->ha, lladdr, dev->addr_len);
114.            neigh_update_hhs(neigh);
115.            if (!(new & NUD_CONNECTED))
116.                neigh->confirmed = jiffies -
117.                          (neigh->parms->base_reachable_time << 1);
118.    #ifdef CONFIG_ARPD
119.            notify = 1;
120.    #endif
121.        }
122.        if (new == old)
123.            goto out;
124.        if (new & NUD_CONNECTED)
125.            neigh_connect(neigh);
126.        else
127.            neigh_suspect(neigh);
128.        if (!(old & NUD_VALID)) {
129.            struct sk_buff *skb;
130.
131.            /* Again: avoid dead loop if something went wrong */
132.
133.            while (neigh->nud_state & NUD_VALID &&
134.                   (skb = __skb_dequeue(&neigh->arp_queue)) != NULL) {
135.                struct neighbour *n1 = neigh;
136.
```

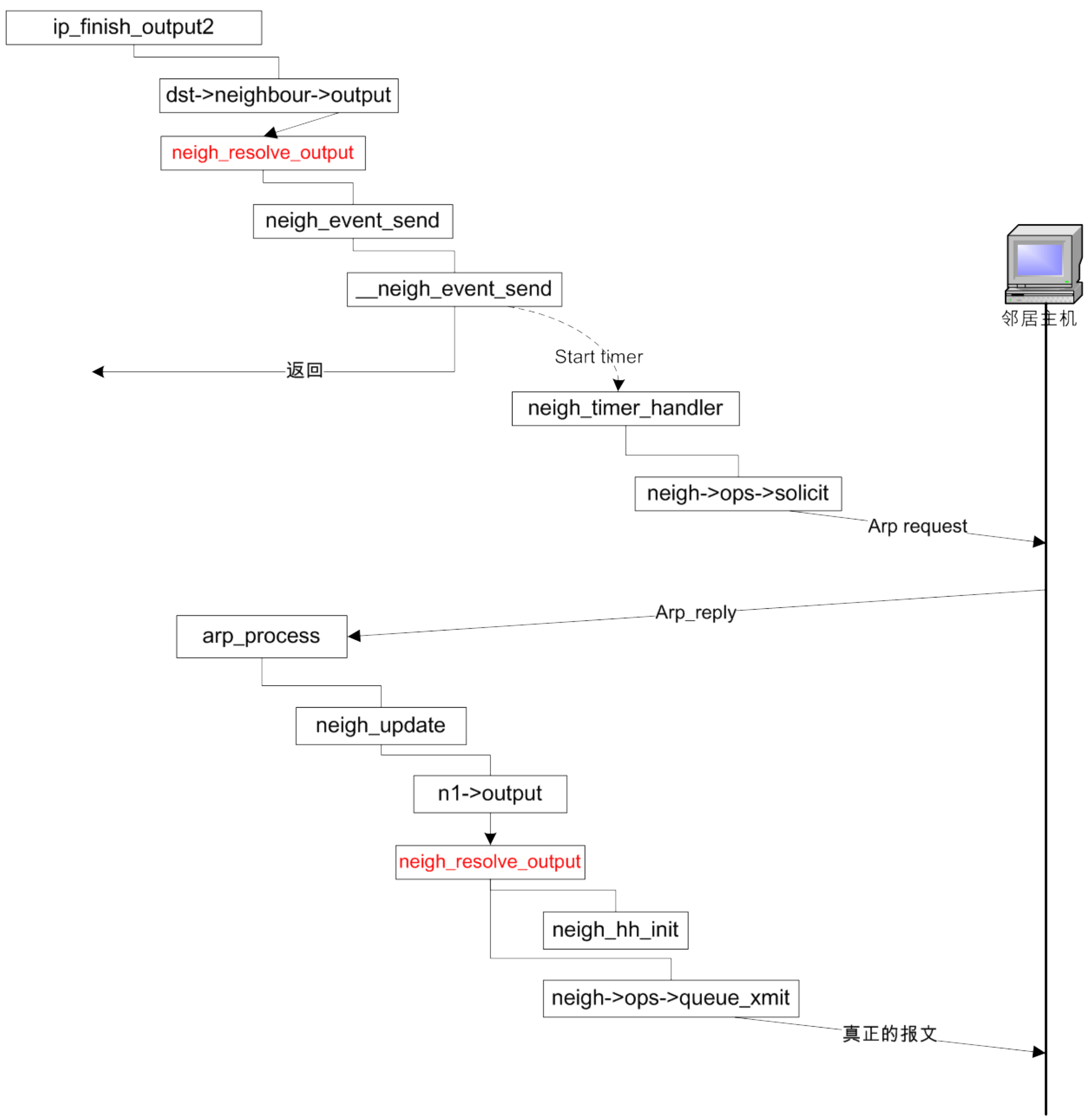
```
137.                /* On shaper/eql skb->dst->neighbour != neigh :( */
138.                if (skb->dst && skb->dst->neighbour)
139.                    n1 = skb->dst->neighbour;
            这时调用 neigh_resolve_output，这已经是第二次调用了。
140.                n1->output(skb);
141.            }
142.            skb_queue_purge(&neigh->arp_queue);
143.        }
144.    out:
145.        if (update_isrouter) {
146.            neigh->flags = (flags & NEIGH_UPDATE_F_ISROUTER) ?
147.                (neigh->flags | NTF_ROUTER) :
148.                (neigh->flags & ~NTF_ROUTER);
149.        }
150.    #ifdef CONFIG_ARPD
151.        if (notify && neigh->parms->app_probes)
152.            neigh_app_notify(neigh);
153.    #endif
154.        return err;
155.    }
```

代码段 4-38 neigh_update 函数

其中调用了 neigh_suspect：

```
1.  static void neigh_suspect(struct neighbour *neigh)
2.  {
3.      struct hh_cache *hh;
4.  
5.      NEIGH_PRINTK2("neigh %p is suspected.\n", neigh);
6.  
7.      neigh->output = neigh->ops->output;
8.  
9.      for (hh = neigh->hh; hh; hh = hh->hh_next)
10.         hh->hh_output = neigh->ops->output;
11. }
```

代码段 4-39 neigh_suspect 函数



图表 4-16 ping 操作的基本流程

4.8 到达设备驱动层

4.8.1 数据链路层帧格式

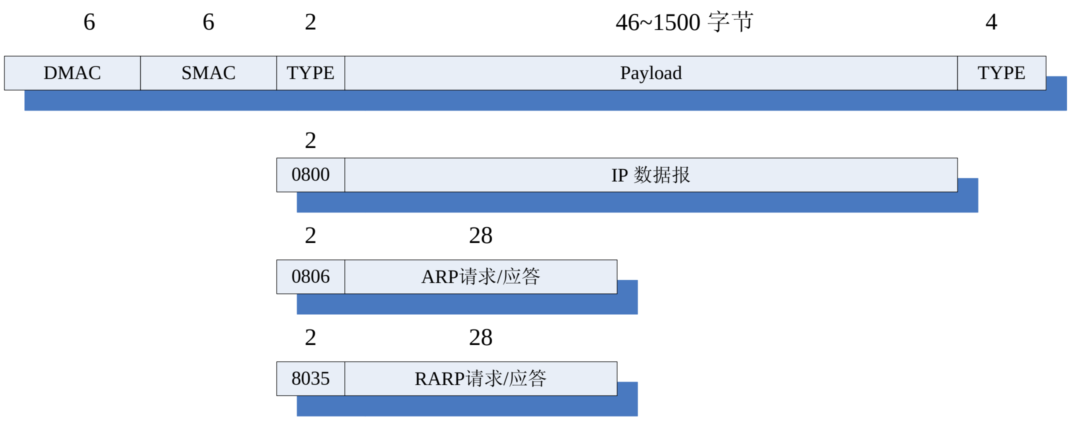
按功能和兼容性划这些数据帧格式对应的协议列举如下：

表格 4-2

Number	Topic
802.1	LAN 结构和概述
802.2	逻辑链路控制

802.3	以太网
802.4	令牌总线
802.5	令牌环
802.7	关于宽带技术的技术顾问组
802.8	关于光纤技术的技术顾问组
802.9	同步 LANs（实时应用）
802.10	虚拟 LANs 和安全
802.11	无线 LANs

环回接口的邻居状态为 NUD_NOARP，属于 NUC_CONNECTED 的子集，所以，hh_output 指向 ops->hh_output，hh 生成后，目的入口 dst_entry 的成员 hh 也指向这个 hh，下一次在发送数据报时，发送函数 ip_finish_output2 会判断目的入口的 hh 成员不为 NULL，所以直接把 dst_entry->hh->hh_data 拷贝成以太网头，然后调用 hh_output(实际函数为 dev_queue_xmit)发送数据报，不再需要构建以太网头。



图表 4-17以太网层数据报文格式

下面这个函数是设备抽象层提供的发送报文的函数。IP 层或者其他更低层的协议都可以使用这个函数发送报文，因为它是设备无关的。

```
1.
2.  /**
3.   *  dev_queue_xmit - transmit a buffer
4.   *  @skb: buffer to transmit
5.   *
6.   *  Queue a buffer for transmission to a network device. The caller must
7.   *  have set the device and priority and built the buffer before calling
8.   *  this function. The function can be called from an interrupt.
9.   *
10.  *  A negative errno code is returned on a failure. A success does not
11.  *  guarantee the frame will be transmitted as it may be dropped due
12.  *  to congestion or traffic shaping.
13.  *
14.  *
    ----------------------------------------------------------------------------
    -------
15.  *     I notice this method can also return errors from the queue disciplines,
16.  *     including NET_XMIT_DROP, which is a positive value.  So, errors can also
17.  *     be positive.
18.  *
19.  *     Regardless of the return value, the skb is consumed, so it is currently
20.  *     difficult to retry a send to this method.  (You can bump the ref count
21.  *     before sending to hold a reference for retry if you are careful.)
22.  *
23.  *     When calling this method, interrupts MUST be enabled.  This is because
```

```
24.  *     the BH enable code must have IRQs enabled so that it will not deadlock.
25.  *          --BLG
26.  */
27.
28. int dev_queue_xmit(struct sk_buff *skb)
29. {
30.     struct net_device *dev = skb->dev;
31.     struct Qdisc *q;
32.     int rc = -ENOMEM;
33.
34. /* GSO will handle the following emulations directly. */
35.     if (netif_needs_gso(dev, skb))
36.         goto gso;
37.
38.     if (skb_shinfo(skb)->frag_list &&
39.         !(dev->features & NETIF_F_FRAGLIST) &&
40.         __skb_linearize(skb))
41.         goto out_kfree_skb;
42.
43. /* Fragmented skb is linearized if device does not support SG,
44.  * or if at least one of fragments is in highmem and device
45.  * does not support DMA from it.
46.  */
47.     if (skb_shinfo(skb)->nr_frags &&
48.         (!(dev->features & NETIF_F_SG) || illegal_highdma(dev, skb)) &&
49.         __skb_linearize(skb))
50.         goto out_kfree_skb;
51.
52. /* If packet is not checksummed and device does not support
53.  * checksumming for this protocol, complete checksumming here.
54.  */
55.     if (skb->ip_summed == CHECKSUM_HW &&
56.         (!(dev->features & NETIF_F_GEN_CSUM) &&
57.          (!(dev->features & NETIF_F_IP_CSUM) ||
58.           skb->protocol != htons(ETH_P_IP))))
59.             if (skb_checksum_help(skb, 0))
60.                 goto out_kfree_skb;
61.
62. gso:
63.     spin_lock_prefetch(&dev->queue_lock);
64.
65. /* 下面代码要关闭软中断，并且停止抢占
66.  */
67.     rcu_read_lock_bh();
68.
69. /* Updates of qdisc are serialized by queue_lock.
70.  * The struct Qdisc which is pointed to by qdisc is now a
71.  * rcu structure - it may be accessed without acquiring
72.  * a lock (but the structure may be stale.) The freeing of the
73.  * qdisc will be deferred until it's known that there are no
74.  * more references to it.
75.  *
76.  * If the qdisc has an enqueue function, we still need to
77.  * hold the queue_lock before calling it, since queue_lock
78.  * also serializes access to the device queue.
79.  */
80.
81.     q = rcu_dereference(dev->qdisc);
82. #ifdef CONFIG_NET_CLS_ACT
83.     skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
84. #endif
85.     if (q->enqueue) {
86.         /* Grab device queue */
87.         spin_lock(&dev->queue_lock);
88.
89.         rc = q->enqueue(skb, q);
90.
91.         qdisc_run(dev);
92.
```

```
93.         spin_unlock(&dev->queue_lock);
94.         rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
95.         goto out;
96.     }
97.
98. /* The device has no queue. Common case for software devices:
99.    loopback, all the sorts of tunnels...
100.
101.           Really, it is unlikely that netif_tx_lock protection is necessary
102.           here.  (f.e. loopback and IP tunnels are clean ignoring statistics
103.           counters.)
104.           However, it is possible, that they rely on protection
105.           made by us here.
106.
107.           Check this and shot the lock. It is not prone from deadlocks.
108.           Either shot noqueue qdisc, it is even simpler 8)
109.          */
110.         if (dev->flags & IFF_UP) {
111.             int cpu = smp_processor_id(); /* ok because BHs are off */
112.
113.             if (dev->xmit_lock_owner != cpu) {
114.
115.                 HARD_TX_LOCK(dev, cpu);
116.
117.                 if (!netif_queue_stopped(dev)) {
118.                     rc = 0;
119.                     if (!dev_hard_start_xmit(skb, dev)) {
120.                         HARD_TX_UNLOCK(dev);
121.                         goto out;
122.                     }
123.                 }
124.
125.             }
126.              ......
127.         }
128.
129.         rc = -ENETDOWN;
130.
131.     out_kfree_skb:
132.         kfree_skb(skb);
133.         return rc;
134.     out:
135.         rcu_read_unlock_bh();
136.         return rc;
137.     }
```

代码段 4-40 dev_queue_xmit 函数

dev_hard_start_xmit 里面比较简单，调用其参数 dev 所指向的 hard_start_xmit 函数指针。

4.8.2 Loopback 设备的发送过程

我们注意到 loopback 设备在初始化的时候挂接了 loopback_dev，并且把实际发送函数定义为这样：.hard_start_xmit = loopback_xmit。即 dev_hard_start_xmit 调用了 loopback_xmit 函数。

那么前面一节提到的 dev→ hard_start_xmit 就是是 loopback_xmit 函数

```
1.  /*
2.   * The higher levels take care of making this non-reentrant (it's
3.   * called with bh's disabled).
4.   */
5.  static int loopback_xmit(struct sk_buff *skb, struct net_device *dev)
6.  {
7.      struct net_device_stats *lb_stats;
8.
9.      skb_orphan(skb);
10.
11.     skb->protocol = eth_type_trans(skb,dev);
12.     skb->dev = dev;
```

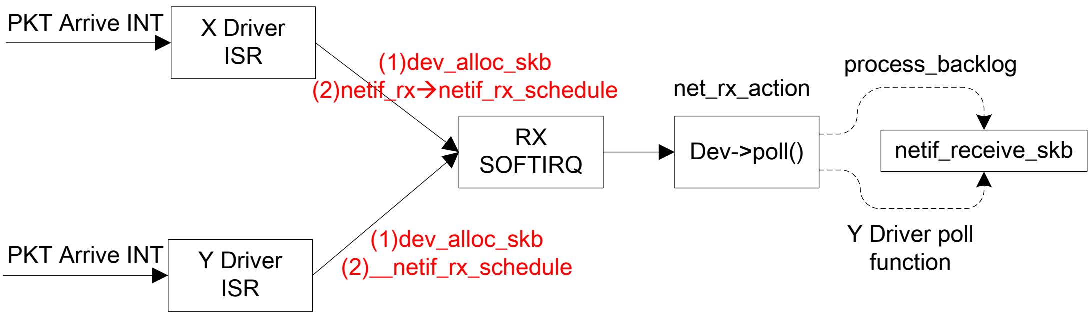
```
13.
14.     ......
15.     dev->last_rx = jiffies;
16.
17.     lb_stats = &per_cpu(loopback_stats, get_cpu());
18.     lb_stats->rx_bytes += skb->len;
19.     lb_stats->tx_bytes = lb_stats->rx_bytes;
20.     lb_stats->rx_packets++;
21.     lb_stats->tx_packets = lb_stats->rx_packets;
        报文直接发送到接收函数中了
22.     netif_rx(skb);
23.
24.     return(0);
25. }
```

代码段 4-41 loopback_xmit 函数

这段代码中，报文没有直接发到设备，而是直接传给了 netif_rx 函数，这是什么意思？好啦，读者们，我们将要拉开接收过程分析的大幕了！

4.9 从中断到路由系统

先来看这么一副图，PKT Arrive INT 表示报文到达 CPU 的中断产生了，某设备驱动的中断服务例程 ISR 于是处理这个中断：



图表 4-18 不同类型的驱动程序造成不同的报文接收方式

这副图告诉我们 Linux 下设备是如何处理接收报文的，其步骤如下：

1. 当中断到来后 ISR 响应，判断是否报文接收中断，如果是那么必定完成如下工作：

   skb = dev_alloc_skb(…);

   skb→protocol = eth_type_trans(skb, dev);

2. 触发软中断，ISR 返回。
   a) 如果设备采用 backlog_dev 设备作为自己的接收后台（如 X driver），那么通过 netif_rx 函数触发软中断，它内部调用 netif_rx_schedule→__netif_rx_schedule
   b) 如果设备没有采用 backlog_dev 设备（如 Y driver），那么开发者必须调用 __netif_rx_schedule 触发软中断
3. 软中断会调用设备定义 poll 函数，此时也有 2 种路径：
   a) 如果设备采用 backlog_dev 设备作为自己的接收后台（如 X driver），那么 poll 就是 process_backlog 函数，它内部调用 netif_receive_skb 函数
   b) 如果设备没有采用 backlog_dev 设备（如 Y driver），那么开发者必须实现相应的 poll 函数，其中必须调用 netif_receive_skb 函数
4. netif_receive_skb 会调用 deliver_skb 将报文传给正确的协议处理函数

下面我们细细分析 netif_rx 函数。为了 CPU 的效率，上层的处理函数的将采用软中断的方式激活，netif_rx 的一个重要工作就是将传入的 sk_buff 放到等候队列中，并置软中断标志位，然后便可放心返回，等待下一次网络数据包的到来

过了一段时间后，一次 CPU 调度会由于某些原因会发生（如某进程的时间片用完）。在进程调度函数即 schedule()中，会检查有没有软中断发生，若有则运行相应的处理函数

```
1.  /**
2.   *  netif_rx    -   传递报文数据到网络处理代码
3.   *  此函数从设备驱动程序中收到一个报文，然后把它放入上层协议的队列中等待处理，它几乎总是成
    功的。
4.   *   该报文可能由于拥塞控制而被丢弃，也可能被上层协议丢弃
5.   *
6.   *  return values:
7.   *  NET_RX_SUCCESS (没有拥塞)
8.   *  NET_RX_CN_LOW   (低拥塞)
9.   *  NET_RX_CN_MOD   (中等程度的拥塞)
10.  *  NET_RX_CN_HIGH  (严重拥塞)
11.  *  NET_RX_DROP    (报文被丢弃)
12.  */
13. int netif_rx(struct sk_buff *skb)
14. {
15.    int this_cpu;
16.    struct softnet_data *queue;
17.    unsigned long flags;
18.
19.    ......
20.
21.    if (!skb->stamp.tv_sec)
22.        net_timestamp(&skb->stamp);
23.
24. /*
25.  * The code is rearranged so that the path is the most
26.  * short when CPU is congested, but is still operating.
27.  */
28.    local_irq_save(flags);
29.    this_cpu = smp_processor_id();
30.    queue = &__get_cpu_var(softnet_data);
31.
32.    __get_cpu_var(netdev_rx_stat).total++;
33.    if (queue->input_pkt_queue.qlen <= netdev_max_backlog)
34.    {
      这个队列中的 skb 还不超长，所以可以进入此判断。
      下面几行代码的含义要分 2 种情况：
      第一种：网口收到第一个报文时，先把接收软中断打开，然后把报文放在队列中，返回。当软中断可
      以执行时就去处理这些报文
      第二种：当网口收到报文后发现队列中还有报文，那么先把此次收到的报文挂到队列中，然后就返回，
      由于队列中还有报文，相关软中断可以在后台继续处理这些报文
35.        if (queue->input_pkt_queue.qlen)
36.        {
37. enqueue:
38.            ②__skb_queue_tail(&queue->input_pkt_queue, skb);
39.             local_irq_restore(flags);
40.            return NET_RX_SUCCESS;
41.        }
42.
43.        ①netif_rx_schedule(&queue->backlog_dev);//在此函数中发起软中断
44.        goto enqueue;
45.    }
46.
47. drop:
48.    ......
49.
50.    kfree_skb(skb);
```

```
51.     return NET_RX_DROP;
52. }
```

代码段 4-42 netif_rx 函数

读者一定要体会上面代码中 39～50 行的技巧，其要表达的意义如下：

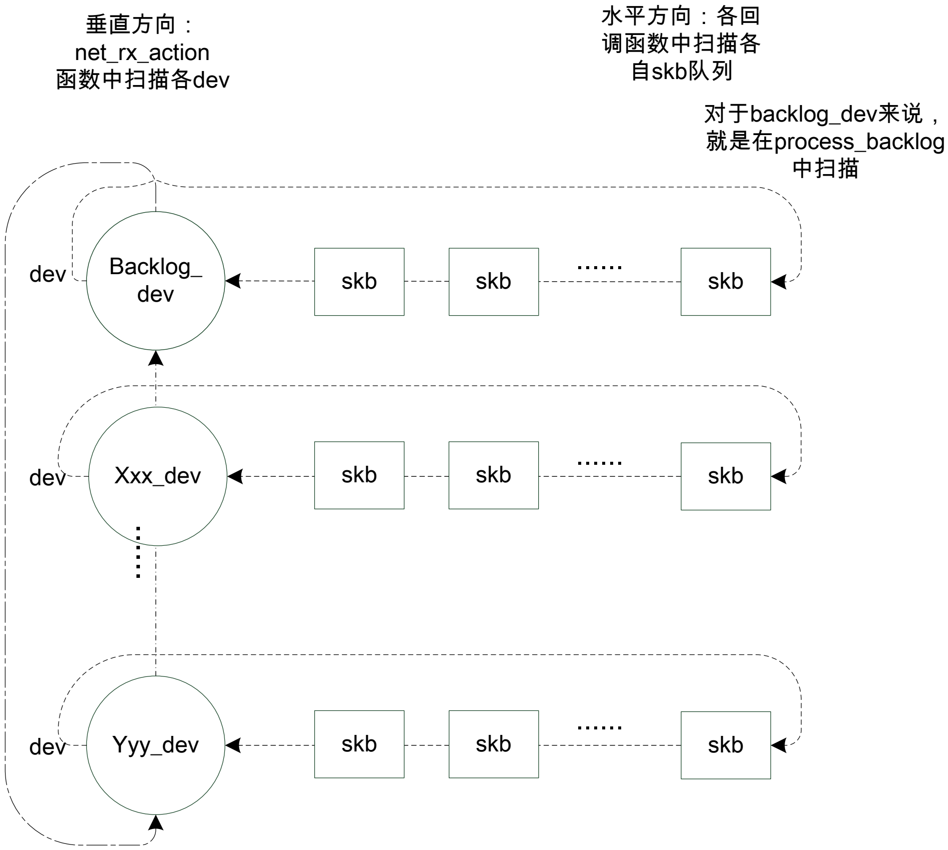
```
1.   if(queue->input_pkt_queue.qlen== 0)//很重要，如果去掉此行代码会有什么结果？
2.   {
3.       ①netif_rx_schedule(&queue->backlog_dev);
4.   }
5.       ②__skb_queue_tail(&queue->input_pkt_queue, skb);
6.
7.   return NET_RX_SUCCESS;
```

代码段 4-43 报文接收队列的等价代码

为什么只有在 input_pkt_queue.qlen 等于 0 的时候发出软中断？由于 net_rx_action 函数与 __netif_rx_schedule 的耦合度很大，它们都操作一个 softnet_data.poll_list 链表，并且不区分是那个设备挂到了链表上，那我们来模拟一下如果把那句判断语句去掉会出现什么情况：

1. 假设input_pkt_queue.qlen为 0，先执行①，然后执行②，似乎没有问题，返回NET_RX_SUCCESS。
2. 当又来一个报文（此时软中断还没有执行机会），于是又执行①，这样softnet_data.poll_list上又多了一个dev，似乎也没有问题。然后再执行②，input_pkt_queue链表上有 2 个报文了。
3. 终于轮到软中断——net_rx_action 执行了，它先取出一个 dev（即 back_log），然后执行其 poll（即 process_backlog）函数，此 process_log 函数扫描 input_pkt_queue 队列，发现有 2 个报文，依次处理，完毕退出到 net_rx_action 函数中
4. net_rx_action 还要扫描 softnet_data.poll_list，取出第二个 dev，还是 back_log，再次执行其 poll 函数，但是 input_pkt_queue 队列中已经没有报文了，于是退出。问题似乎有，但不大，因为只是浪费一次循环。
5. 如果我们再假设突然来 100 个报文，那么无效循环可以达到 100 次，是不是呢？

其实这也不是什么不能解决的问题，只要在__netif_rx_schedule 中加入对同一 dev 的检查，不过，不管用什么样的算法来检查，都要浪费时间，因为这可是在中断里面。至于 Linux 内核为什么要写成要 goto 的语句，我没看懂。读者可以帮我想一下。



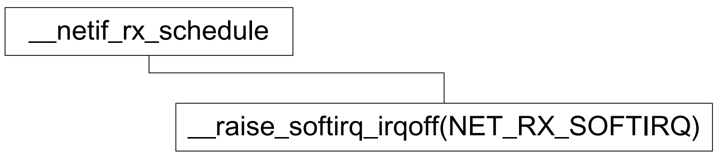
代码段 4-44 不同的接收设备使用不同的接收队列

说完 netif_rx 的内部流程，我们再看看 RX SOFTIRQ 是如何产生作用的。

首先是它如何被注册到系统软中断体制上的。在网络设备初始化例程那一章节中我们看到 net_dev_init 函数。这个函数挂接了 2 各软中断，一个是 TX SOFTIRQ，一个是 RX SOFTIRQ，后者就是接收软中断，此软中断的处理函数被定为 net_rx_action：

open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);

另一方面，驱动程序的 ISR 也要激发一个软中断，不管是使用封装好的 netif_rx，还是直接调用__netif_rx_schedule，其内部通过__raise_softirq_irqoff 来标记一个软中断的产生：



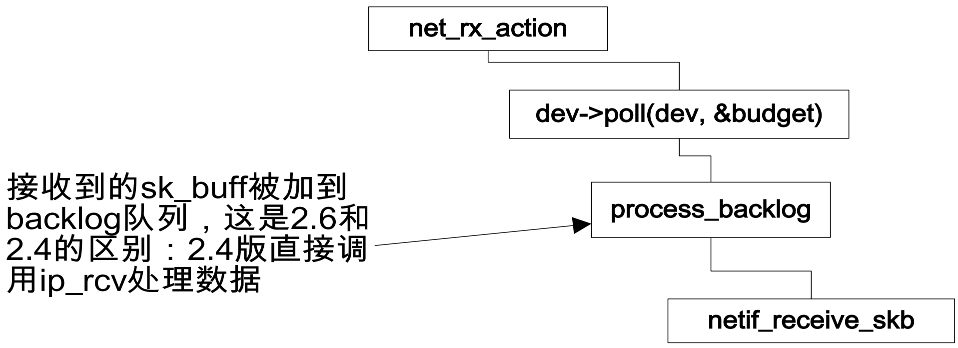
图表 4-19 __netif_rx_schedule 函数调用树

前面已经说过软中断被处理的时机，一是在中断发生的时候，二是在系统发生进程调度的时刻。当可以处理软中断的时候，系统会检查是否发生 NET_RX_SOFTIRQ 软中断，若有则调用 net_rx_action。在 net_rx_action 中：

dev = list_entry(queue->poll_list.next,   struct net_device, poll_list);

如果系统此 dev 就是 softnet_data 中的 backlog_dev，它的 poll 函数指针在 net_dev_init 中被赋

予 process_backlog，此函数检查 softnet_data→input_pkt_queue 是否有数据，如果有的话就调用 netif_receive_skb。



图表 4-20 net_rx_actiion 函数调用树

- 通过 packet_ptype_base[16]这个数组解决。这个数组包含了需要接收包的协议，以及它们的接收函数的入口。
- ip 协议接收数据是通过 ip_rcv，arp 协议是通过 arp_rcv。
- 如果有协议想把自己添加到这个数组，是通过 dev_add_pack 实现。Ip 层的注册是在 ip_init 中

```
1.  int netif_receive_skb(struct sk_buff *skb)
2.  {
3.      struct packet_type *ptype, *pt_prev;
4.      int ret = NET_RX_DROP;
5.      unsigned short type;
6.  
7.  #ifdef CONFIG_NETPOLL_RX
8.      if (skb->dev->netpoll_rx && skb->dev->poll && netpoll_rx(skb)) {
9.          kfree_skb(skb);
10.         return NET_RX_DROP;
11.     }
12. #endif
13. 
14.     if (!skb->stamp.tv_sec)
15.         net_timestamp(&skb->stamp);
16. 
17.     skb_bond(skb);
18. 
19.     __get_cpu_var(netdev_rx_stat).total++;
20. 
21. #ifdef CONFIG_NET_FASTROUTE
22.     if (skb->pkt_type == PACKET_FASTROUTE) {
23.         __get_cpu_var(netdev_rx_stat).fastroute_deferred_out++;
24.         return dev_queue_xmit(skb);
25.     }
26. #endif
27. 
28.     skb->h.raw = skb->nh.raw = skb->data;
29.     skb->mac_len = skb->nh.raw - skb->mac.raw;
30. 
31.     pt_prev = NULL;
```

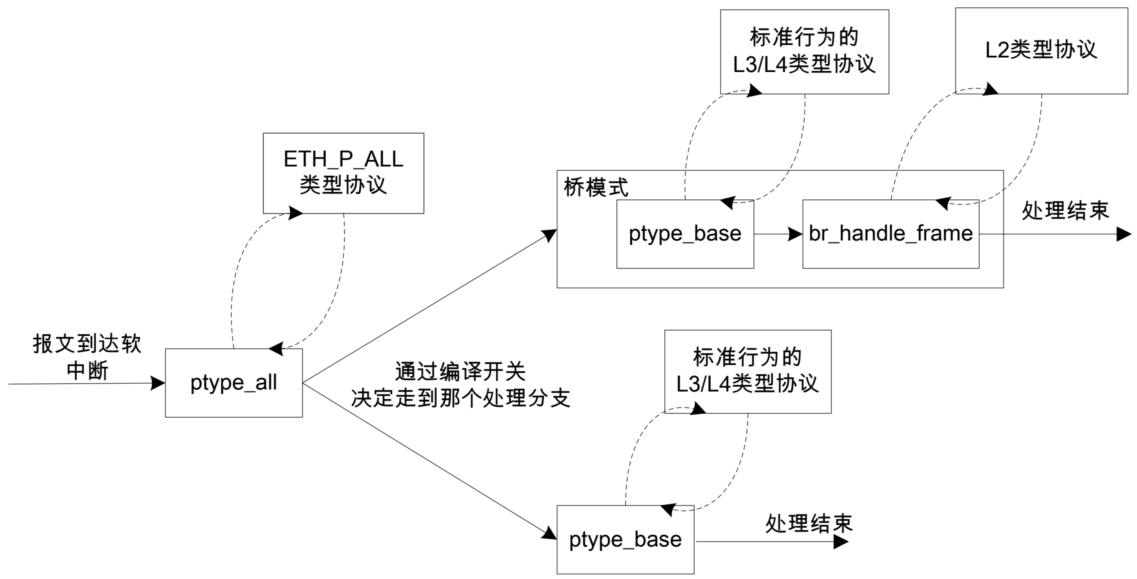
如果设置了将报文全部收上去，那么会执行下面的代码，要注意的是，收上去之后的报文还需要第 40 行之后的代码处理，也就是说，此时收上去的报文不仅仅是给 IP 协议栈用的，而是给诸如防火墙、某个特权用户甚至无聊的黑客使用的。(好可怕！☺)

```
32.     list_for_each_entry_rcu(ptype, &ptype_all, list) {
33.         if (!ptype->dev || ptype->dev == skb->dev) {
34.             if (pt_prev)
35.                 ret = deliver_skb(skb, pt_prev, 0);
36.             pt_prev = ptype;
```

```
           这里要注意的是最后一个 ptype 没有被执行，它被放到第 49 行执行
37.         }
38.     }
39.
40.     handle_diverter(skb);
     如果配置了桥模式 BRIDGE， 那么__handle_bridge 函数将会有具体的形式，我们将会在第 7 章介绍
     这个函数，如果没有配置 BRIDGE，那么这个函数是空定义。那么 pt_prev 还是等于 NULL。
41.     if (__handle_bridge(skb, &pt_prev, &ret))
42.         goto out;
43.
44.     type = skb->protocol;
     在内核代码中，ptype_all 和 ptype_base 是处于互补的关系，参见 dev_add_pack 函数。
45.     list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
46.         if (ptype->type == type &&
47.             (!ptype->dev || ptype->dev == skb->dev)) {
             如果执行 ptype_all 中有钩子函数，那么必然执行这个函数，然后才把当前的 ptype 传给
             pt_prev，然后才开始执行 ptype_base 中的函数
48.             if (pt_prev)
49.                 ret = deliver_skb(skb, pt_prev, 0);
50.             pt_prev = ptype;
                ptype_base 中最后一个成员必定在第 54 行执行
51.         }
52.     }
     在这里开始调正常报文的处理函数，比如 IP 包或 ARP 包（当然，如果 ptype_base 中没有指定处理函
     数，那么这里执行的还是 ptype_all 中的处理函数）——这一段代码比较烦琐，解释起来也比较累！
53.     if (pt_prev) {
54.         ret = pt_prev->func(skb, skb->dev, pt_prev);
55.     } else {
56.         kfree_skb(skb);
57.         ret = NET_RX_DROP;
58.     }
59. out:
60.     return ret;
61. }
```

代码段 4-45 netif_receive_skb 函数

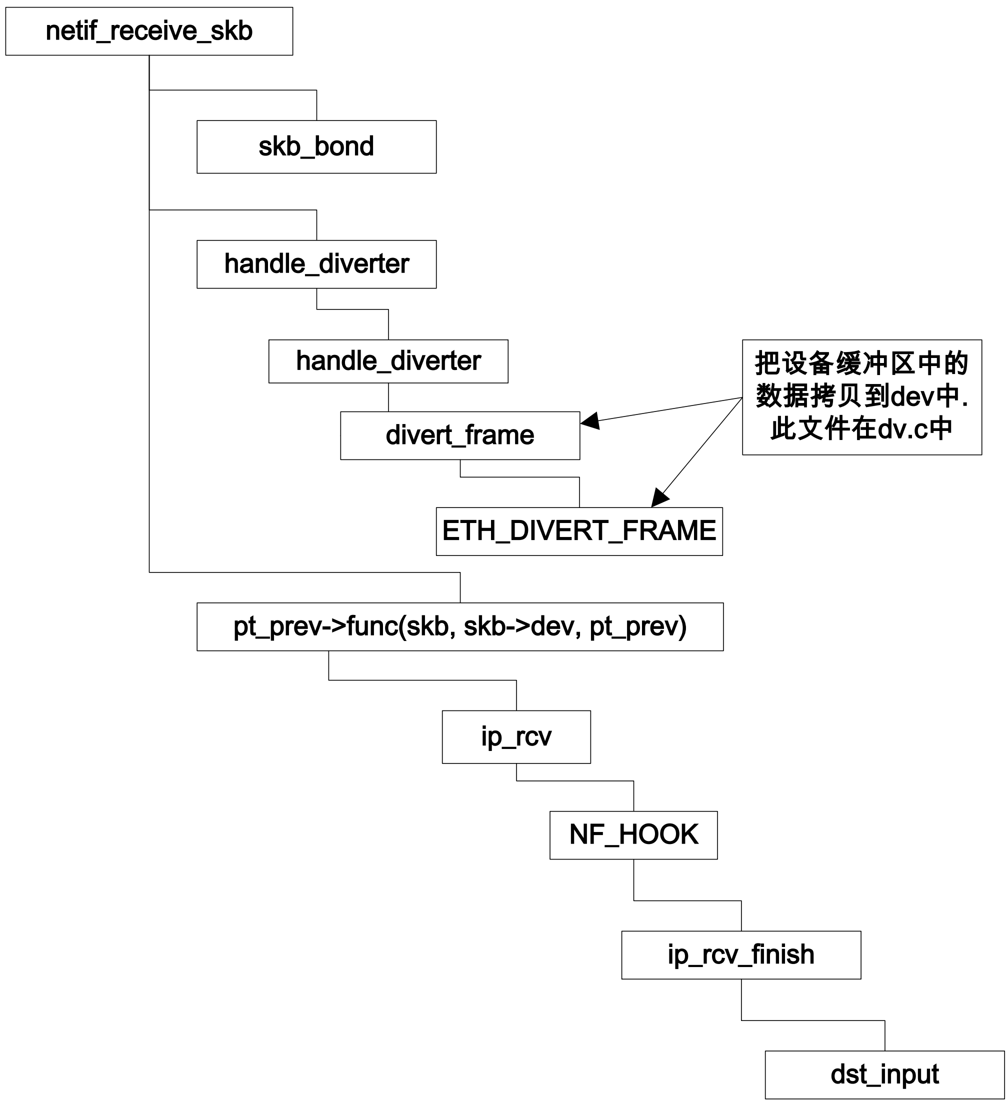
这段代码的基本思路如下，一个报文要经过层层“过滤”才能结束其生命，“过滤器”就是每个协议的处理函数：



图表 4-21 报文到达被不同层次协议处理的原理图

在这里要提醒中国的读者，特别是英语不太灵光的读者，看源代码能提高自己的英语水准。forward 和 delive 在英文词典中意思类似，但是，从代码上就可以看出两者之间的区别：前者是水平方向的“转发”，后者是向上的“转发”，以后我们写类似的代码时，最好能借鉴这里的经验。

我们在之前的初始化中看到 ptype_base 目前只有 2 个成员，一个是 arp_rcv 函数，一个是 ip_rcv 函数。我们已经讨论过 arp_rcv 了，下面讨论 ip_rcv。这个函数的前面大多是相关于 ip 协议的检查，包括 ip 头的检查，我们跳过这部分检查内容。函数最后通过 NF_HOOK 宏转入 ip_rcv_finish()函数



图表 4-22 netif_receive_skb 函数调用树

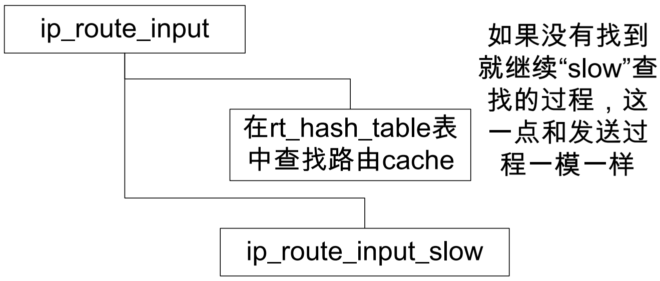
因为数据包是刚接收到的还没有设置关于 dst_entry 结构的路由信息，那么首先是设置数据包的路由，这是通过 ip_route_input()函数来实现的。下面是 ip_rcv_finish 的代码:

```
1.  static inline int ip_rcv_finish(struct sk_buff *skb)
2.  {
3.      struct net_device *dev = skb->dev;
4.      struct iphdr *iph = skb->nh.iph;
5.
6.  /*
7.   *  Initialise the virtual path cache for the packet. It describes
8.   *  how the packet travels inside Linux networking.
```

```
9.   */
10.
11.    if (skb->dst == NULL) {
12.        /*路由为空的时候检查路由，因为要确定这个包是怎么处理的，是转发还是本地分发*/
13.        if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
14.            goto drop;
15.    }
16.
17.    if (iph->ihl > 5) {
18.        struct ip_options *opt;
19.
20.     /* It looks as overkill, because not all
21.        IP options require packet mangling.
22.        But it is the easiest for now, especially taking
23.        into account that combination of IP options
24.        and running sniffer is extremely rare condition.
25.                                      --ANK (980813)
26.     */
27.
28.        if (skb_cow(skb, skb_headroom(skb))) {
29.            IP_INC_STATS_BH(IpInDiscards);
30.            goto drop;
31.        }
32.        iph = skb->nh.iph;
33.
34.        if (ip_options_compile(NULL, skb))
35.            goto inhdr_error;
36.
37.        opt = &(IPCB(skb)->opt);
38.        if (opt->srr) {
39.            struct in_device *in_dev = in_dev_get(dev);
40.            if (in_dev) {
41.                if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
42.                    if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
43.                     /*source route option */
44.                    in_dev_put(in_dev);
45.                    goto drop;
46.                }
47.             }
48.            if (ip_options_rcv_srr(skb)) /*源路由选项处理*/
49.                goto drop;
50.        }
51.    }
52.
53.    return dst_input(skb);
54.
55. inhdr_error:
56.    IP_INC_STATS_BH(IpInHdrErrors);
57. drop:
58.    kfree_skb(skb);
59.    return NET_RX_DROP;
60. }
```

代码段 4-46 ip_rcv_finish 函数

INET 域中有两个地方需要查询输入路由，一个是当收到一个 IP 数据报，ip_rcv 将其交给 ip_rcv_finish 后，ip_rcv_finish 判断 skb->dst 是否为 NULL(因为对于环回接口上收到的数据报，其 dst 是存在的，不需要查询输入路由)，如果为 NULL，则需要查询得到输入路由。另一个地方是当收到一个 ARP 数据报，arp_rcv 将其交给 arp_process 处理时，arp_process 也需要查询得到该 skb 的输入路由。



图表 4-23ip_route_input 函数调用树

```
1  /*
2   *    NOTE. We drop all the packets that has local source
3   *    addresses, because every properly looped back packet
4   *    must have correct destination already attached by output routine.
5   *
6   *    Such approach solves two big problems:
7   *    1. Not simplex devices are handled properly.
8   *    2. IP spoofing attempts are filtered with 100% of guarantee.
9   */
10
11 static int ip_route_input_slow(struct sk_buff *skb, u32 daddr, u32 saddr,
12                u8 tos, struct net_device *dev)
13 {
14     struct fib_result res;
15     struct in_device *in_dev = in_dev_get(dev);
16     struct flowi fl = { .nl_u = { .ip4_u =
17                   { .daddr = daddr,
18                  .saddr = saddr,
19                  .tos = tos,
20                  .scope = RT_SCOPE_UNIVERSE,
21 #ifdef CONFIG_IP_ROUTE_FWMARK
22                  .fwmark = skb->nfmark
23 #endif
24                   } },
25              .iif = dev->ifindex };
26      unsigned flags = 0;
27      u32      itag = 0;
28      struct rtable * rth;
29      unsigned hash;
30      u32      spec_dst;
31      int      err = -EINVAL;
32      int      free_res = 0;
33
34  /* 如果在这个接口上关掉 IP 使能，那么必须退出*/
35      if (!in_dev)
36          goto out;
37
38  /* Check for the most weird martians, which can be not detected
39     by fib_lookup.
40   */
41      if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
42          goto martian_source;
43
44      if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
45          goto brd_input;
46
47  /* Accept zero addresses only to limited broadcast;
48   * I even do not know to fix it or not. Waiting for complains :-)
49   */
50      if (ZERONET(saddr))
51          goto martian_source;
52
```

```
53      if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
54          goto martian_destination;
55
56  /*
57   *  Now we are ready to route packet.
58   */
```
现在要到路由系统（即 FIB 表）中查找对应路由了，这个步骤和 ip_route_output_slow 中一样
```
59      if ((err = fib_lookup(&fl, &res)) != 0) {
60          if (!IN_DEV_FORWARD(in_dev))
61              goto e_hostunreach;
62          goto no_route;
63      }
64      free_res = 1;
65
66      if (res.type == RTN_BROADCAST)
67          goto brd_input;
68
69      if (res.type == RTN_LOCAL) {
70          int result;
71          result = fib_validate_source(saddr, daddr, tos,
72                           loopback_dev.ifindex,
73                           dev, &spec_dst, &itag);
74          if (result < 0)
75              goto martian_source;
76          if (result)
77              flags |= RTCF_DIRECTSRC;
78          spec_dst = daddr;
```
发现是目的地是本地地址，跳转到相应代码，注意，路由插入的代码也是在其后
```
79          goto local_input;
80       }
81
82      if (!IN_DEV_FORWARD(in_dev))
83          goto e_hostunreach;
84      if (res.type != RTN_UNICAST)
85          goto martian_destination;
```
将查找到的路由插入到路由 cache 中，这也和 ip_route_output_slow 保持一致
```
86      err = ip_mkroute_input(skb, &res, &fl, in_dev, daddr, saddr, tos);
87      ......
88
89    done:
90      if (free_res)
91          fib_res_put(&res);
```
对于目的地不是本地的报文，从这里就退出了
```
92 out: return err;
93
94 brd_input:
95      if (skb->protocol != htons(ETH_P_IP))
96          goto e_inval;
97
98      if (ZERONET(saddr))
99          spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
100      else {
101          err = fib_validate_source(saddr, 0, tos, 0, dev, &spec_dst,
102                           &itag);
103          if (err < 0)
104              goto martian_source;
105          if (err)
106              flags |= RTCF_DIRECTSRC;
107      }
108      flags |= RTCF_BROADCAST;
109      res.type = RTN_BROADCAST;
110      RT_CACHE_STAT_INC(in_brd);
```

如果目的地是本地地址，如:127.0.0.1 或我们之前配置的 192.168.18.1 进入下面的代码
```
111 local_input:
```
这下面的代码和前面调用的 ip_mkroute_input 里面完成的工作几乎一样,除了指定 dst.input 函数，
```
112      rth = dst_alloc(&ipv4_dst_ops);
113
114      rth->u.dst.output= ip_rt_bug;
```

```
115
116     atomic_set(&rth->u.dst.__refcnt, 1);
117     rth->u.dst.flags= DST_HOST;
118     if (in_dev->cnf.no_policy)
119         rth->u.dst.flags |= DST_NOPOLICY;
120     rth->fl.fl4_dst = daddr;
121     rth->rt_dst = daddr;
122     rth->fl.fl4_tos = tos;
123 #ifdef CONFIG_IP_ROUTE_FWMARK
124     rth->fl.fl4_fwmark= skb->nfmark;
125 #endif
126     rth->fl.fl4_src = saddr;
127     rth->rt_src = saddr;
128 #ifdef CONFIG_NET_CLS_ROUTE
129     rth->u.dst.tclassid = itag;
130 #endif
```

下面的代码就是这样写的，是不是比较怪异啊？

```
131     rth->rt_iif =
132     rth->fl.iif = dev->ifindex;
133     rth->u.dst.dev  = &loopback_dev;
134
135     rth->idev   = in_dev_get(rth->u.dst.dev);
136     rth->rt_gateway = daddr;
137     rth->rt_spec_dst= spec_dst;
```

下面这个函数我们会在后面的接收过程中再次看到

```
138     rth->u.dst.input= ip_local_deliver;
139     rth->rt_flags   = flags|RTCF_LOCAL;
140     if (res.type == RTN_UNREACHABLE) {
141         rth->u.dst.input= ip_error;
142         rth->u.dst.error= -err;
143         rth->rt_flags   &= ~RTCF_LOCAL;
144     }
145     rth->rt_type    = res.type;
146     hash = rt_hash_code(daddr, saddr ^ (fl.iif << 5));
147     err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);
148     goto done;
149
150 no_route:
151     RT_CACHE_STAT_INC(in_no_route);
152     spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
153     res.type = RTN_UNREACHABLE;
154     goto local_input;
155
156     /*
157      *  Do not cache martian addresses: they should be logged (RFC1812)
158      */
159 martian_destination:
160     RT_CACHE_STAT_INC(in_martian_dst);
161
162 e_hostunreach:
163        err = -EHOSTUNREACH;
164        goto done;
165
166 e_inval:
167     err = -EINVAL;
168     goto done;
169
170 e_nobufs:
171     err = -ENOBUFS;
172     goto done;
173
174 martian_source:
175     ip_handle_martian_source(dev, in_dev, skb, daddr, saddr);
176     goto e_inval;
177 }
```

代码段 4-47 ip_route_input_slow 函数

此函数调用__mkroute_input 去增加一条路由 cache，其中的步骤和__mkroute_output 几乎同出一辙。

```
178 static inline int __mkroute_input(struct sk_buff *skb,
179                  struct fib_result* res,
180                  struct in_device *in_dev,
181                  u32 daddr, u32 saddr, u32 tos,
182                  struct rtable **result)
183 {
184
185     struct rtable *rth;
186     int err;
187     struct in_device *out_dev;
188     unsigned flags = 0;
189     u32 spec_dst, itag;
190
191     /* get a working reference to the output device */
192     out_dev = in_dev_get(FIB_RES_DEV(*res));
193     if (out_dev == NULL) {
194         ......
195         return -EINVAL;
196     }
197
198
199     err = fib_validate_source(saddr, daddr, tos, FIB_RES_OIF(*res),
200                  in_dev->dev, &spec_dst, &itag);
201     if (err < 0) {
202         ip_handle_martian_source(in_dev->dev, in_dev, skb, daddr,
203                      saddr);
204
205         err = -EINVAL;
206         goto cleanup;
207     }
208
209     if (err)
210         flags |= RTCF_DIRECTSRC;
211
212     if (out_dev == in_dev && err && !(flags & (RTCF_NAT | RTCF_MASQ)) &&
213         (IN_DEV_SHARED_MEDIA(out_dev) ||
214          inet_addr_onlink(out_dev, saddr, FIB_RES_GW(*res))))
215         flags |= RTCF_DOREDIRECT;
216
217     if (skb->protocol != htons(ETH_P_IP)) {
218         /* Not IP (i.e. ARP). Do not create route, if it is
219          * invalid for proxy arp. DNAT routes are always valid.
220          */
221         if (out_dev == in_dev && !(flags & RTCF_DNAT)) {
222             err = -EINVAL;
223             goto cleanup;
224         }
225     }
226
227 创建一条路由 cache，并指定了相关操作
228     rth = dst_alloc(&ipv4_dst_ops);
229     if (!rth) {
230         err = -ENOBUFS;
231         goto cleanup;
232     }
233     rth->u.dst.flags= DST_HOST;
234 #ifdef CONFIG_IP_ROUTE_MULTIPATH_CACHED
235     if (res->fi->fib_nhs > 1)
236         rth->u.dst.flags |= DST_BALANCED;
237 #endif
238     if (in_dev->cnf.no_policy)
239         rth->u.dst.flags |= DST_NOPOLICY;
240     if (in_dev->cnf.no_xfrm)
241         rth->u.dst.flags |= DST_NOXFRM;
242     rth->fl.fl4_dst = daddr;
243     rth->rt_dst = daddr;
```

```
244     rth->fl.fl4_tos = tos;
245     rth->fl.fl4_src = saddr;
246     rth->rt_src = saddr;
247     rth->rt_gateway = daddr;
248     rth->rt_iif     =
249         rth->fl.iif = in_dev->dev->ifindex;
250     rth->u.dst.dev  = (out_dev)->dev;
251     rth->idev   = in_dev_get(rth->u.dst.dev);
252     rth->fl.oif     = 0;
253     rth->rt_spec_dst= spec_dst;
254 
255     rth->u.dst.input = ip_forward;
256     rth->u.dst.output = ip_output;
257 
258     rt_set_nexthop(rth, res, itag);
259 
260     rth->rt_flags = flags;
261 
262     *result = rth;
263     err = 0;
264  cleanup:
265     /* release the working reference to the output device */
266     return err;
267 }
```

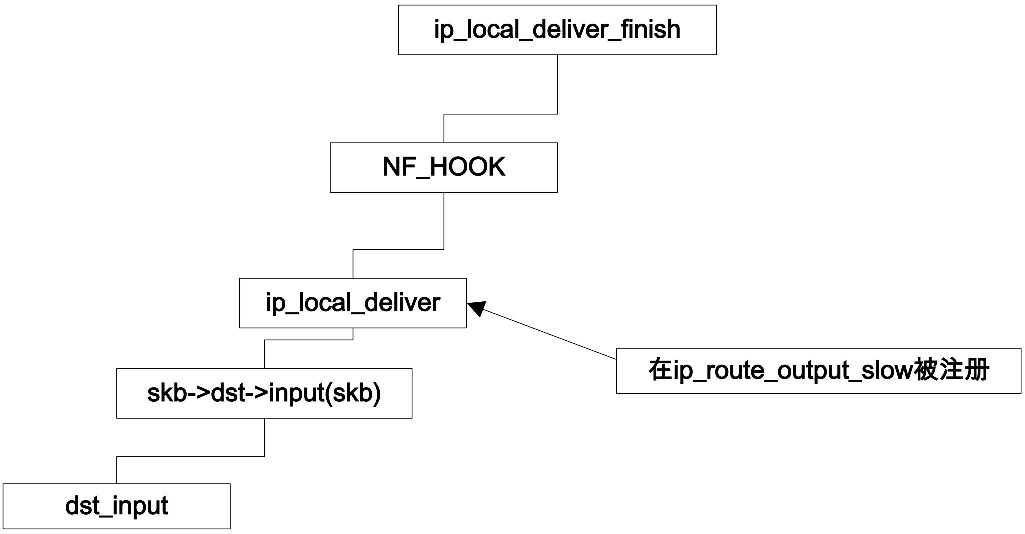
代码段 4-48 __mkroute_input 函数

输入路由查询的调用接口是 ip_route_input，该函数首先从路由缓存表 rt_hash_table 中查找，如果缓存表中不存在，则调用函数 ip_route_input_slow 生成一个新的输入路由项，ip_route_input_slow 对输入路由分多种情况处理，如果 skb 的目的地址是广播地址，则调用 dst_alloc 创建本地输入的路由项，struct rtable 的 rt_flags 成员的值为 RTCF_DIRECTSRC | RTCF_BROADCAST | RTCF_LOCAL；如果 skb 的目的地址是本地接收地址，则创建本地输入的路由项，struct rtable 的 rt_flags 成员的值为 RTCF_DIRECTSRC | RTCF_LOCAL。

如果收到的 skb 不是本地接收的，是需要进行转发的，并且该网络设备接口也确实允许转发，则调用 ip_mkroute_input 创建路由项。

重点注意这里是将路由表中的进入路径钩子挂入了 ip_local_deliver 函数。然后将新建立的路由表设置进入数据包的路由表指针，并且放入全局的路由表缓存 rt_hash_table［］中，我们继续看 ip_rcv_finish()函数的代码，设置了数据包的路由表后，最后进入 dst_input()函数。

其中 dst_input 的调用关系如下图，注意此图是反的，表示数据是从底层往上层送：



图表 4-24 dst_input 函数调用树

skb->dst-input 在 ip_route_output_slow 函数中定义。如果是发送到本机的包，往下的操作函数是 ip_local_deliver，如果这条路由是用来转发的，对应操作函数是 ip_forward，如果查找不到路由的处理情况则是 ip_error。

下面就是 ip_local_deliver 函数，首先是对到来的数据包进行碎片重组的检查和处理，最后转入了 ip_local_deliver_finish()函数

```
1.  /*
2.   *  Deliver IP Packets to the higher protocol layers.
3.   */
4.  int ip_local_deliver(struct sk_buff *skb)
5.  {
6.  /*
7.   *  Reassemble IP fragments.
8.   */
9.
10.    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {/*判断是否是一个分片包*/
11.        skb = ip_defrag(skb); /*分片重组*/
12.        if (!skb)
13.            return 0;
14.    }
15.
16.    return ip_local_deliver_finish (skb);
17. }
```

代码段 4-49 ip_local_deliver 函数

注意在 Kernel2.4 内核代码中，p_run_ipprot 函数变成了 ip_local_deliver_finish，其代码如下：

```
1.  static inline int ip_local_deliver_finish(struct sk_buff *skb)
2.  {
3.     int ihl = skb->nh.iph->ihl*4;
4.
5.     __skb_pull(skb, ihl);
6.
7.     /* Point into the IP datagram, just past the header. */
8.     skb->h.raw = skb->data;
9.
10.    rcu_read_lock();
11.    {
12.     /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
13.        int protocol = skb->nh.iph->protocol;
```

```
14.         int hash;
15.         struct sock *raw_sk;
16.         struct inet_protocol *ipprot;
17.
18. resubmit:
       对每个报文要先检查是否是 RAW 报文,检查的方式是从 raw_v4_hash 表中查找是否有对应的 RAW,
       在 inet_create 函数中 RAW socket 曾经调用了自己 hash 函数，将自己挂在了此 hash 表中。
       然而不是 RAW 的 socket 不会往此 hash 表中加数据。
19.         hash = protocol & (MAX_INET_PROTOS - 1);
20.         raw_sk = sk_head(&raw_v4_htable[hash]);
21.
22.      向上层分发收到的目的地址是本机的包,首先分发原始套接字，注意：由于 ping 程序本身是 raw 应
     用，所以会进入到 raw_v4_input 中，但是实际并没有通过这个函数将报文收到应用层，为什么呢？
     请参考该函数的讲解，而对于其它 RAW IP 的应用，比如 OSPF 就会通过此函数把报文发送到用户层
23.         if (raw_sk && !raw_v4_input(skb, skb->nh.iph, hash))
24.             raw_sk = NULL;
25.     /*没有冲突的哈希表，直接定向到协议指针*/
26.         if ((ipprot = inet_protos[hash]) != NULL)
27.         {
28.             int ret;
     对应的操作函数是 tcp_v4_rcv,icmp_rcv,igmp_rcv,udp_rcv，在本书目前的例子中，由于是
     ping 应用程序，那么它应该是 icmp_rcv，
29.             ret = ipprot->handler(skb);
30.              IP_INC_STATS_BH(IpInDelivers);
31.         }
32.          else { 如果没有对应的高层协议处理此报文，要么发送目标不可达，要么释放该报文
33.             if (!raw_sk) {
34.                icmp_send(skb, ICMP_DEST_UNREACH,
35.                          ICMP_PROT_UNREACH, 0);
36.                 }
37.             } else
38.                 kfree_skb(skb);
39.         }
40.     }
41.  out:
42.
43.     return 0;
44. }
```

代码段 4-50 ip_local_deliver_finish 函数

这里值得总结一下。当 IP 报文从软中断上来的时候，必须从找到 ptype_base[]数组中找到对应函数，目前内核中与 IP 有关的只有 3 个，一个是 ip_packet_type，另一个是 arp_packet_type，最后是 rarp_packet_type。如果是 ip 报文，当执行到 ip_local_deliver_finish 时，则应该从 inet_protos[]数组中找到对应函数，里面包含 ICMP、UDP、TCP 等上层协议的处理函数。这 2 个数组极易造成混淆，因为都有一个回调函数。在经历这么多函数的研究，读者可能都有点晕了。一个比较好的记忆方法是：对于前一个，其回调函数是 func，处理 packet，后一个回调函数是 handler，处理协议。

4.10 ICMP

IP 协议并不是一个可靠的协议，它不保证数据被送达，那么，自然的，保证数据送达的工作应该由其他的模块来完成。其中一个重要的模块就是 ICMP(网络控制报文)协议。它是 TCP/IP 协议集中的一个子协议,属于网络层协议,主要用于在主机与路由器之间传递控制信息,包括报告错误、交换受限控制和状态信息等。

当传送 IP 数据包发生错误－－比如主机不可达，路由不可达等等，ICMP 协议将会把错误信

息封包，然后传送回给主机。给主机一个处理错误的机会，这 也就是为什么说建立在 IP 层以上的协议是可能做到安全的原因。ICMP 数据包由 8bit 的错误类型和 8bit 的代码和 16bit 的校验和组成。而前 16bit 就组成了 ICMP 所要传递的信息。书上的图 6－3 清楚的给出了错误类型和代码的组合代表的意思。从技术角度来说，ICMP 就是一个“错误侦测与回报机制”，其目的就是让我们能够检测网路的连线状况，也能确保连线的准确性，其功能主要有：

· 侦测远端主机是否存在。
· 建立及维护路由资料。
· 重导资料传送路径。
· 资料流量控制。

ICMP 属于 TCP/I P 协议族的一个部分，但从原理上来说它并不是不可缺少的，因为协议栈没有它照样可以跑起来，但由于它主要用在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等，所以如果没有它我们几乎不能检测协议栈的正常运转，。比如 ping 就是使用 ICMP 协议的例子，还有我们常说的 traceroute 应用程序。ICMP 传递差错报文以及其他需要注意的信息。ICMP 报文通常被 IP 层或更高层协议（ TCP 或 UDP）使用。一些 ICMP 报文把差错报文返回给用户进程。

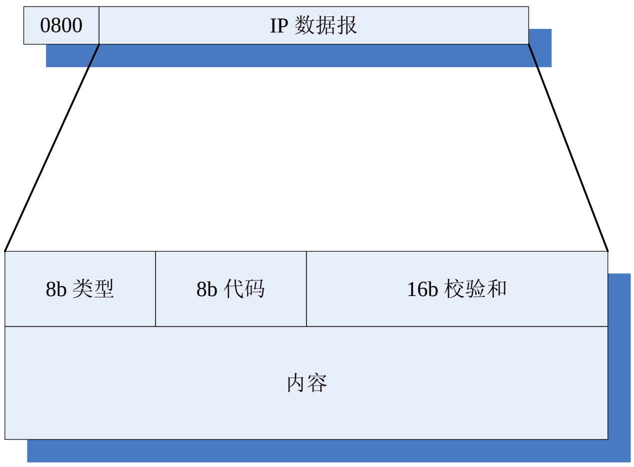
ICMP 协议大致分为两类，一种是查询报文，一种是差错报文。其中查询报文有以下几种用途:

1. ping 查询（不要告诉我你不知道 ping 程序）
2. 子网掩码查询（用于无盘工作站在初始化自身的时候初始化子网掩码）
3. 时间戳查询（可以用来同步时间）

而差错报文则产生在数据传送发生错误的时候。就不赘述了。

4.10.1 ICMP 报文格式

ICMP 报文是在 I P 数据报内部被传输的，如下图所示。ICMP 的正式规范参见 RFC 792 [Posterl1 9 8 1 b]。我们常说的 TCP 和 UDP 能承载数据，但 ICMP 仅包含控制信息，如下图所示。ICMP 不象 TCP 或 UDP 有端口，但它确实含有两个域：类型(type)和代码(code)。ICMP 在沟通之中，主要是透过不同的类别(Type)与代码(Code) 让机器来识别不同的连线状况。

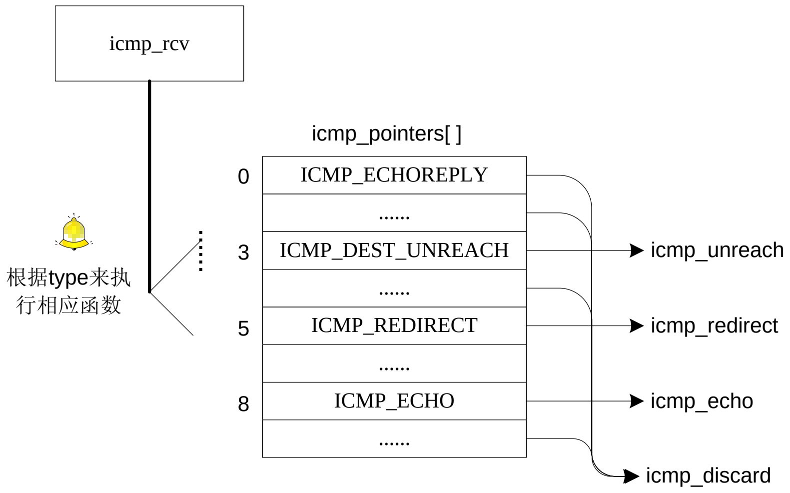


图表 4-25ICMP 数据报文格式

ICMP 报文的格式如图 6 - 2 所示。所有报文的前 4 个字节都是一样的，但是剩下的其他字节则互不相同。下面我们将逐个介绍各种报文格式。

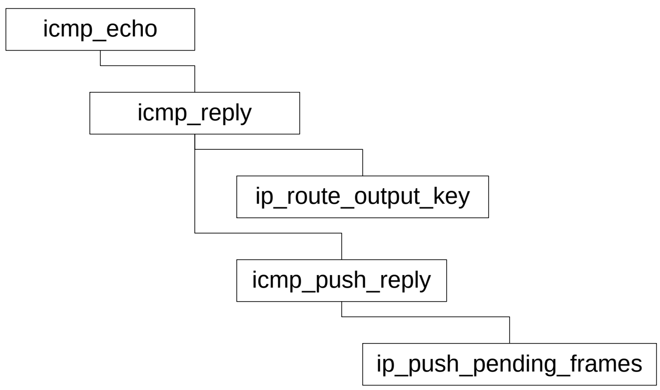
类型字段可以有 1 5 个不同的值，以描述特定类型的 ICMP 报文。某些 ICMP 报文还使用代码字段的值来进一步描述不同的条件。

检验和字段覆盖整个 ICMP 报文。ICMP 的检验和是必需的。



图表 4-26 icmp_rcv 处理接收到的消息

当报文到达 ip_local_deliver_finish 时，发现是 ICMP 数据报文类型，那么会执行 icmp_rcv，它会根据 ICMP 的类型来执行相应的函数。这些函数都放到一个 icmp_points[]指针数组中，大部分指针都指向 icmp_discard，里面什么都不做，连个 return 语句都没有。我们比较常见的就是上图中列出的几个函数。



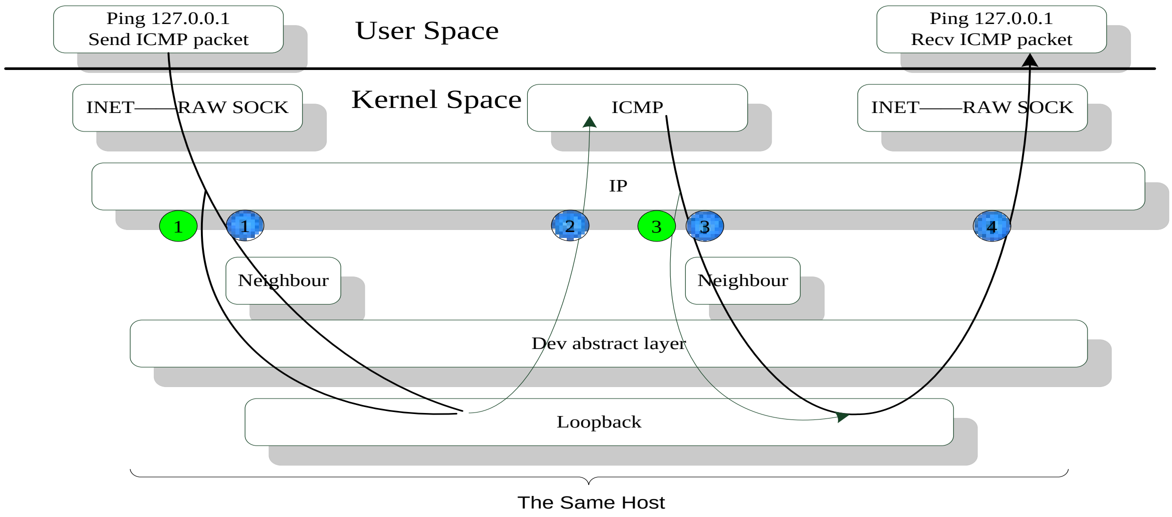
图表 4-27 icmp_echo 函数调用树

从上图中可以看出来，icmp 是基于 IP 的应用，它首先去查询路由表，然后才能把报文发出来。当分组被发给错误的路由器时，产生重定向报文。该路由器把分组转发给正确的路由器，并发回一个 I C M P 重定向报文，系统把信息记入它自己的路由表。

4.10.2 ping 本机地址及回环地址

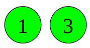
Ping 命令利用 ICMP 回射请求报文和回射应答报文来测试目标系统是否可达。

ICMP 请求和 ICMP 应答报文是配合工作的。当源主机向目标主机发送了 ICMP 请求数据包后，它期待着目标主机的回答。目标主机在收到一个 ICMP 回射请求数据包后，它会交换源、目的主机的地址，然后将收到的 ICMP 请求数据包中的数据部分原封不动地封装在自己的 ICMP 回射应答数据包中，然后发回给发送 ICMP 请求的一方。如果校验正确，发送者便认为目标主机的服务正常，也即物理连接畅通。



图表 4-28协议栈的交互——目的地址是本机

——neigh_resolve_output

——dev_queue_xmit

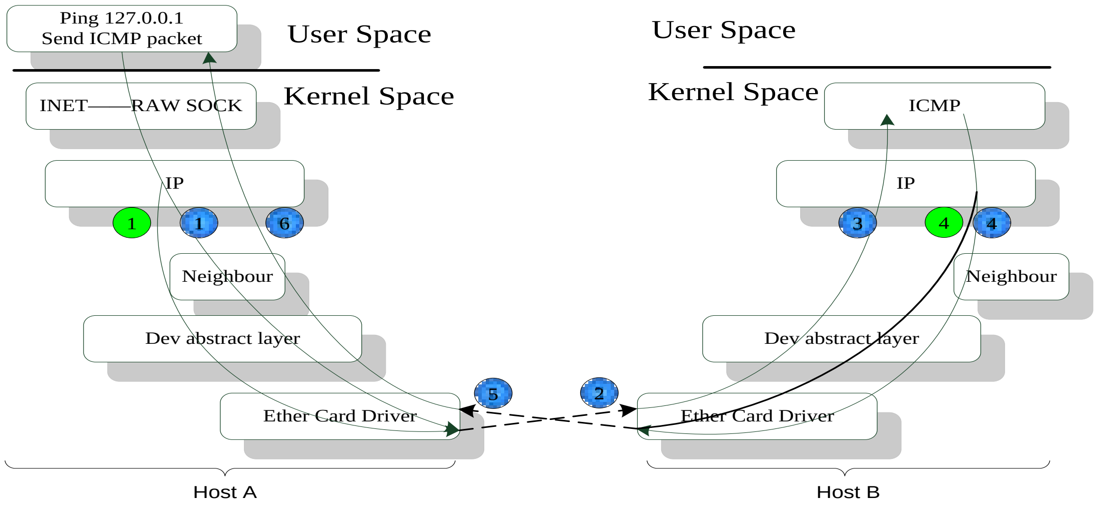
——ip_local_deliver

当收到来自对端主机的 icmp 回显应答时，ip_local_deliver_finish 函数会先检查哈希表 raw_v4_htable。因为在创建 socket 时，inet_create 会把协议号 IPPROTO_ICMP 的值赋给 socket 的成员 num，并以 num 为键值，把 socket 存入哈项表 raw_v4_htable，raw_v4_htable[IPPROTO_ICMP&(MAX_INET_PROTOS-1)]上即存放了这个 socket，实际上是一个 socket 的链表，如果其它还有 socket 要处理这个回显应答，也会被放到这里，组成一个链表，ip_local_deliver_finish 收到数据报后，取出这个 socket 链表(目前实际上只有一项)，调用 raw_v4_input，把 skb 交给每一个 socket 进行处理。然后，还需要把数据报交给 inet_protos[IPPROTO_ICMP&(MAX_INET_PROTOS-1)]，即 icmp_rcv 处理，因为对于 icmp 报文，每一个都是需要经过协议栈处理的，但对回显应答，icmp_rcv 只是简单丢弃，并未实际处理。

到这里报文上到了 IP 层，也经过了 ICMP 协议的处理，然后报文会交给 INET 层，不过我们会在“从内核到用户”一节中讲解。

4.10.3 ping 外部地址

到现在为止，我们研究的是目的地址是本机的 ping 过程，下面我们要介绍如果目的地址不是本机的情况。既然我们已经给出了 ping 本机的内部过程，那下图就是目的地址是外部主机的情况之一——目的地主机（在图中是 Host B）收到了 ICMP 报文。



图表 4-29协议栈的交互——目的地址是直连主机

注意这里是 2 台主机 A 和 B。

1 4 ——neigh_resolve_output

1 4 ——dev_queue_xmit

3 6 ——ip_local_deliver

可以得出结论，当源、目的地主机直接相连的时候，我们是可以想象其和 ping 本地地址的情况类似，除了我们要注意设备驱动程序发送的接口是特定设备的发送接口而不是 loopback 设备接口的发送。

现在我们还要考虑另外一种情况，就是当某个主机收到一个目的地址不属于自己的报文，它应该如何处理，这个处理过程就是协议栈的网络行为。之前我们讨论的协议栈处理本机报文属于主机行为。我们已经知道在 ip_route_input_slow 函数中，首先调用 fib_lookup 查找相应路由，如果目的地址不是本地，那么就调用 ip_mkroute_input 去指定 input 和 output 函数指针，在最终调用的__mkroute_input 函数中有这样两行代码：

rth->u.dst.input = ip_forward;

rth->u.dst.output = ip_output;

于是分析一下这个我们未遇见过的 ip_forward 函数，下面是经过整理的 ip_forward 代码：

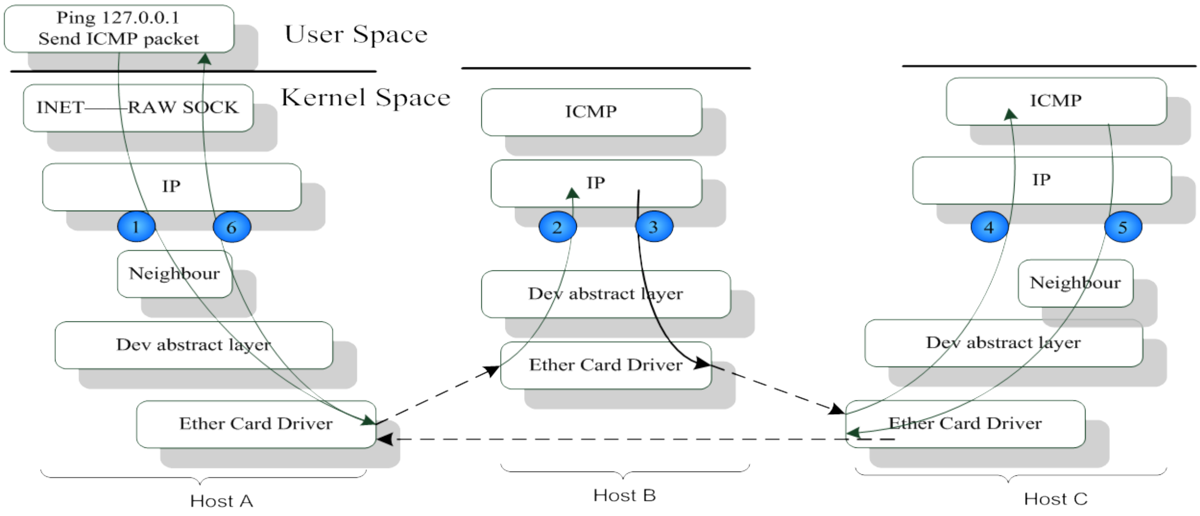
```
1.  int ip_forward(struct sk_buff *skb)
2.  {
3.  struct iphdr *iph; /* Our header */
4.  struct rtable *rt; /* Route we use */
```

```
5.  struct ip_options * opt= &(IPCB(skb)->opt);
6.
7.  if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
8.      return NET_RX_SUCCESS;
    /*广播，多播，混杂模式得到的包不允许转发*/
9.  if (skb->pkt_type != PACKET_HOST)
10.     goto drop;
11.
12. skb->ip_summed = CHECKSUM_NONE;
13.
14. /*
15.  *  According to the RFC, we must first decrease the TTL field. If
16.  *  that reaches zero, we must reply an ICMP control message telling
17.  *  that the packet's lifetime expired.
18.  */
19.
20. iph = skb->nh.iph;
21.
22. if (iph->ttl <= 1)
23.             goto too_many_hops;
24. iph = skb->nh.iph;
25. rt = (struct rtable*)skb->dst;
26.
   /*如果路由选出的源路由包的下一站不是网关，丢弃这个包*/
27. if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
28.     goto sr_failed;
29.
30. /* We are about to mangle packet. Copy it! */
31.     /*重新组织这个 SKBUFF 结构*/
32. if (skb_cow(skb, LL_RESERVED_SPACE(rt->u.dst.dev)+rt->u.dst.header_len))
33.     goto drop;
34. iph = skb->nh.iph;
35.
36. /* Decrease ttl after skb cow done */
37. ip_decrease_ttl(iph); /*减少 ttl，并且重新计算校验和*/
38.
39. /*
40.  *  We now generate an ICMP HOST REDIRECT giving the route
41.  *  we calculated.
42.  */
    /*路由重新定向且没有源路由选项的时候，必须产生一个重定向的 icmp 包*/
43. if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr)
44.     ip_rt_send_redirect(skb);
45.
46. skb->priority = rt_tos2priority(iph->tos);
47.
48. return ip_forward_finish(skb);
49.
50. sr_failed:
51.         /*
52.     *    Strict routing permits no gatewaying
53.     */
54.         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
55.         goto drop;
56.
57. too_many_hops:
58.         /* Tell the sender its packet died... */
59.         icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
60. drop:
61. kfree_skb(skb);
62. return NET_RX_DROP;
63. }
```

代码段 4-51 ip_forward

ip_forward_finish 最后还是调用了 dst_output。我们知道 dst_output 实际上调用 skb->dst->output，于是联系上一页中对这个函数指针的赋值，就知道，它指向了 ip_output。这个

函数我们已经研究过了，这里不多说了。总结如下：



图表 4-30协议栈的交互——目的地址是远端主机

注意这里是 3 台主机。Host B 处于中间位置，数据流没有到达 ICMP 模块，而是到达 IP 层后，发现本机不是目的主机，于是调用 ip_forward 把数据转发出去。

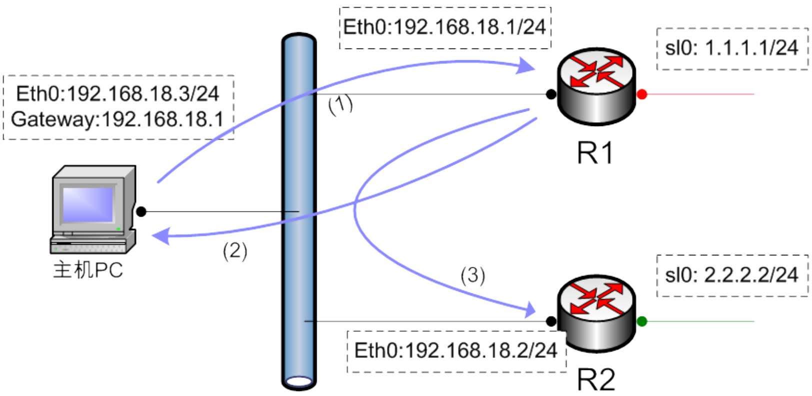
2——ip_forward

3——ip_output

在目的主机是经过中间主机（或路由器）才能到达的情况下，中间主机的 ICMP 模块没有参与到处理这个 ICMP 报文的过程中。

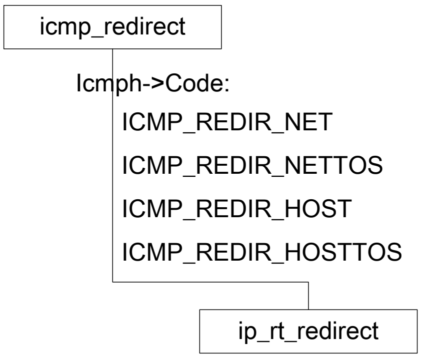
但是还有一种情况，作为中间主机的 ICMP 参与到寻路的过程中，这就是路由重定向。ICMP 虽然不是路由协议，但是有时它也可以指导数据包的流向（使数据流向正确的网关）。ICMP 协议通过 ICMP 重定向数据包（类型 5、代码 0：网络重定向）达到这个目的。

我们看下面这副网络配置图：



图表 4-31 ICMP 重定向示意图

主机 PC 要 ping 路由器 R2 的 sl0 地址：2.2.2.2，主机判断出目的地址和自己的 IP 地址属于不同的网段，因此它要将 ICMP 请求包发往自己的默认网关 192.168.18.1（路由器 R1 的 Eth0 接口）。我们现在假设主机和 R1 之间的 ARP 过程已经完成，路由器 R1 便将此 ICMP 请求转发到路由器 R2 的 Eth0 接口：192.168.18.2（这要求路由器 R1 正确配置了到网络 192.168.18.3/24 的路由）。此外，路由器 R1 还要发送一个 ICMP 重定向消息给主机 PC，通知主机 PC 对于主机 PC 请求的地址的网关是：192.168.18.3。路由器 R2 此时会发送一个 ARP 请求消息请求主机 PC 的 MAC 地址，而主机 PC 会发送 ARP 应答消息给路由器 R2。最后路由器 R2 通过获得的主机 PC 的 MAC 地址信息，将 ICMP 应答消息发送给主机 PC。那么我们下面就看看这个发往 PC 的重定向消息做了什么事：



图表 4-32 ipmp_redirect 函数调用树

当主机收到 ICMP REDIRECT 的时候应该走入下面这个函数：

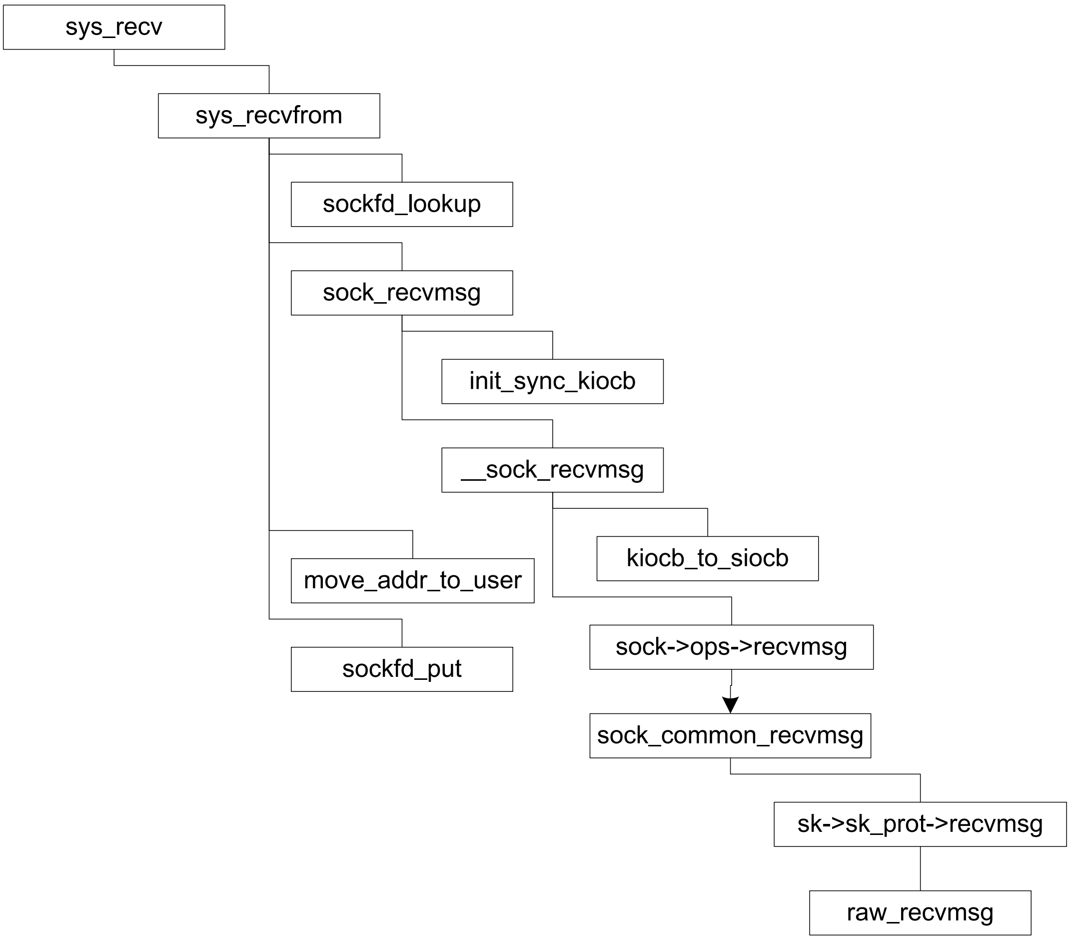
```
1.  void ip_rt_redirect(u32 old_gw, u32 daddr, u32 new_gw,
2.          u32 saddr, struct net_device *dev)
3.  {
4.  int i, k;
5.  struct in_device *in_dev = in_dev_get(dev);
6.  struct rtable *rth, **rthp;
7.  u32  skeys[2] = { saddr, 0 };
8.  int  ikeys[2] = { dev->ifindex, 0 };
9.  struct netevent_redirect netevent;
10.
11. if (new_gw == old_gw || !IN_DEV_RX_REDIRECTS(in_dev)
12.     || MULTICAST(new_gw) || BADCLASS(new_gw) || ZERONET(new_gw))
13.     goto reject_redirect;
14.
15. if (!IN_DEV_SHARED_MEDIA(in_dev)) {
16.     if (!inet_addr_onlink(in_dev, new_gw, old_gw))
17.         goto reject_redirect;
18.     if (IN_DEV_SEC_REDIRECTS(in_dev) && ip_fib_check_default(new_gw, dev))
19.         goto reject_redirect;
20. } else {
21.     if (inet_addr_type(new_gw) != RTN_UNICAST)
22.         goto reject_redirect;
23. }
24.
25. for (i = 0; i < 2; i++) {
26.     for (k = 0; k < 2; k++) {
27.         unsigned hash = rt_hash_code(daddr,
28.                         skeys[i] ^ (ikeys[k] << 5));
29.
30.         rthp=&rt_hash_table[hash].chain;
31.
32.         while ((rth = rcu_dereference(*rthp)) != NULL) {
33.             struct rtable *rt;
34.
```

```
35.             if (rth->fl.fl4_dst != daddr ||
36.                 rth->fl.fl4_src != skeys[i] ||
37.                 rth->fl.oif != ikeys[k] ||
38.                 rth->fl.iif != 0) {
39.                 rthp = &rth->u.rt_next;
40.                 continue;
41.             }
42.
43.             if (rth->rt_dst != daddr ||
44.                 rth->rt_src != saddr ||
45.                 rth->u.dst.error ||
46.                 rth->rt_gateway != old_gw ||
47.                 rth->u.dst.dev != dev)
48.                 break;
49.
50.             rt = dst_alloc(&ipv4_dst_ops);
51.
52.             /* Copy all the information. */
53.             *rt = *rth;
54.                 INIT_RCU_HEAD(&rt->u.dst.rcu_head);
55.             rt->u.dst.__use     = 1;
56.             atomic_set(&rt->u.dst.__refcnt, 1);
57.             rt->u.dst.child     = NULL;
58.
59.             rt->u.dst.obsolete = 0;
60.             rt->u.dst.lastuse  = jiffies;
61.             rt->u.dst.path     = &rt->u.dst;
62.             rt->u.dst.neighbour= NULL;
63.             rt->u.dst.hh       = NULL;
64.             rt->u.dst.xfrm     = NULL;
65.
66.             rt->rt_flags       |= RTCF_REDIRECTED;
67.
68.             /* Gateway is different ... */
69.             rt->rt_gateway     = new_gw;
70.
71.             /* Redirect received -> path was valid */
72.             dst_confirm(&rth->u.dst);
73.
74.             if (arp_bind_neighbour(&rt->u.dst) ||
75.                 !(rt->u.dst.neighbour->nud_state &
76.                     NUD_VALID)) {
77.                 if (rt->u.dst.neighbour)
78.                     neigh_event_send(rt->u.dst.neighbour, NULL);
79.                 rt_drop(rt);
80.                 goto do_next;
81.             }
82.
83.             netevent.old = &rth->u.dst;
84.             netevent.new = &rt->u.dst;
85.             call_netevent_notifiers(NETEVENT_REDIRECT,
86.                             &netevent);
87.
88.             rt_del(hash, rth);
89.             if (!rt_intern_hash(hash, rt, &rt))
90.                 ip_rt_put(rt);
91.             goto do_next;
92.         }
93.         do_next:
94.         ;
95.     }
96. }
97.
98. return;
99.
100.    reject_redirect:
101.
102.    }
```

代码段 4-52 ip_rt_redirect 函数

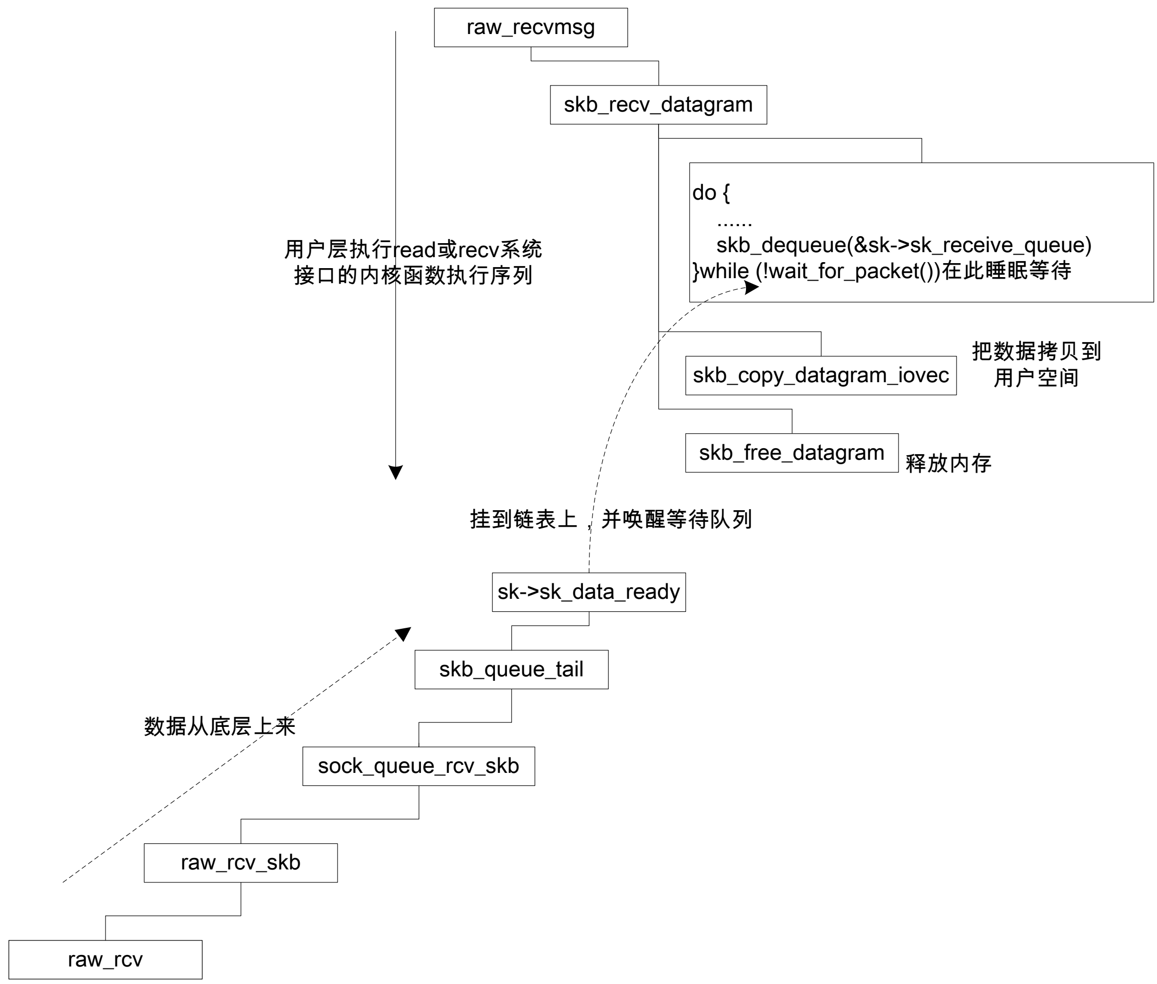
4.11 从内核到用户

而在接收的另一端，用户调用 recvfrom 系统接口等待，其内部操作如下：



图表 4-33 sys_rcv 函数调用树

为了明白数据是如何从协议栈到达用户层的，我们必须从两个方向理解这个过程。



图表 4-34 从两个方向来理解报文是如何到达用户层的

raw_rcv 内部比较简单，无非是把报文挂到队列中，最后感觉好像是 sk→sk_data_ready 唤醒了应用层的等待函数，比如 recvfrom，那么此函数是什么呢？我们现在有必要回顾一下创建 socket 的时候协议栈做了些什么分配。

在创建 socket{}的时候调用此函数：

```
1.  void sock_init_data(struct socket *sock, struct sock *sk)
2.  {
3.  skb_queue_head_init(&sk->sk_receive_queue);
4.  skb_queue_head_init(&sk->sk_write_queue);
5.  skb_queue_head_init(&sk->sk_error_queue);
6.  #ifdef CONFIG_NET_DMA
7.  skb_queue_head_init(&sk->sk_async_wait_queue);
8.  #endif
9.
10. sk->sk_send_head    =   NULL;
11.
12. init_timer(&sk->sk_timer);
13.
14. sk->sk_allocation   =   GFP_KERNEL;
15. sk->sk_rcvbuf       =   sysctl_rmem_default;
16. sk->sk_sndbuf       =   sysctl_wmem_default;
17. sk->sk_state        =   TCP_CLOSE;
18. sk->sk_socket       =   sock;
19.
20. sock_set_flag(sk, SOCK_ZAPPED);
21.
22. if(sock)
```

```
23. {
24.     sk->sk_type =   sock->type;
25.     sk->sk_sleep    =   &sock->wait;
26.     sock->sk    =   sk;
27. } else
28.     sk->sk_sleep    =   NULL;
29.
30. rwlock_init(&sk->sk_dst_lock);
31. rwlock_init(&sk->sk_callback_lock);
32. lockdep_set_class(&sk->sk_callback_lock,
33.            af_callback_keys + sk->sk_family);
34.
35. sk->sk_state_change=   sock_def_wakeup;
36. sk->sk_data_ready  =   sock_def_readable;
37. sk->sk_write_space =   sock_def_write_space;
38. sk->sk_error_report=   sock_def_error_report;
39. sk->sk_destruct    =   sock_def_destruct;
40.
41. sk->sk_sndmsg_page =   NULL;
42. sk->sk_sndmsg_off  =   0;
43.
44. ......
45. sk->sk_write_pending   =   0;
46. sk->sk_rcvlowat    =   1;
47. sk->sk_rcvtimeo    =   MAX_SCHEDULE_TIMEOUT;
48. sk->sk_sndtimeo    =   MAX_SCHEDULE_TIMEOUT;
49.
50. sk->sk_stamp.tv_sec    = -1L;
51. sk->sk_stamp.tv_usec   = -1L;
52.
53. }
```

代码段 4-53 sock_init_data 函数

原来是 sock_def_readable。

```
1.  static void sock_def_readable(struct sock *sk, int len)
2.  {
3.      if (sk->sk_sleep && waitqueue_active(sk->sk_sleep))
4.          wake_up_interruptible(sk->sk_sleep);
5.      sk_wake_async(sk,1,POLL_IN);
6.
7.  }
```

代码段 4-54 sock_def_readable 函数

wake_up_interruptible（）：TASK_INTERRUPTIBLE-> TASK_RUNNING

将 wait_queue 中的所有状态为 TASK_INTERRUPTIBLE 的进程状态都置为 TASK_RUNNING,并将它们都放到 running queue 中去。

现在从另一个方向来看重点来看，当内核处理完 ICMP 报文后，它会直接调用 raw_v4_input，该函数遍历 raw_v4_htable[protocol&(MAX_INET_PROTOS-1)]链表，找出协议号(inet->num)，目的 ip 地址，源 ip 地址，输入设备接口都匹配的 socket，克隆一个 skb 交给它处理。但协议号是 IPPROTO_ICMP(我们当前正处理的情况)时，则还需要判断该 icmp 类型的报文是否是被过滤掉的，如果是，则不处理。但协议栈是永远不会屏蔽回显应答报文的，所以，可以无视这行代码

收到的 skb 交给 raw_rcv 处理，raw_rcv 调用 raw_rcv_skb，把收到的 skb 放入 socket 的接收队列。应用程序收到数据报，解析 icmp 首部即可。

```
1.  /* IP input processing comes here for RAW socket delivery.
2.   * Caller owns SKB, so we must make clones.
3.   *
4.   * RFC 1122: SHOULD pass TOS value up to the transport layer.
5.   * -> It does. And not only TOS, but all IP header.
```

```
6.   */
7.  int raw_v4_input(struct sk_buff *skb, struct iphdr *iph, int hash)
8.  {
9.  struct sock *sk;
10. struct hlist_head *head;
11. int delivered = 0;
12.
13. read_lock(&raw_v4_lock);
14. head = &raw_v4_htable[hash];
15. if (hlist_empty(head))
16.     goto out;
17. sk = __raw_v4_lookup(__sk_head(head), iph->protocol,
18.              iph->saddr, iph->daddr,
19.              skb->dev->ifindex);
20.
21. while (sk) {
22.     delivered = 1;
    下面这个判断足以让我们的 ping 应用程序从这个函数中退出，因为我们的 iph->protocol 是
    IPPROTO_ICMP
23.     if (iph->protocol != IPPROTO_ICMP || !icmp_filter(sk, skb)) {
24.         struct sk_buff *clone = skb_clone(skb, GFP_ATOMIC);
25.
26.         /* Not releasing hash table! */
27.         if (clone)
28.             raw_rcv(sk, clone);
29.     }
30.     sk = __raw_v4_lookup(sk_next(sk), iph->protocol,
31.                  iph->saddr, iph->daddr,
32.                  skb->dev->ifindex);
33. }
34. out:
35. read_unlock(&raw_v4_lock);
36. return delivered;
37. }
```

代码段 4-55 raw_v4_input 函数

第5章　传输层实现的研究

首先要说明的是，我不太想研究这两个协议，因为这不是我兴趣所在，还有就是，我觉得 SCTP 似乎也比较热门，TCP 反而有点日暮西山的感觉了。但前面看到了很多关于 UDP 和 TCP 的初始化代码，不研究这两个传输层协议又有点对不起自己。于是我就挑了一些比较关键的部分说了说，内容上比较跳，注释也不丰富，如果没有一点网络基础的读者还不能完整理解我要说的内容，希望读者原谅。

5.1　进一步到 UDP

关于 IP 层的实现上一节已经介绍了，现在开始介绍 UDP 层，这一层属于传输层应用，UDP 协议基于 IP 层，而 UDP 程序基于 UDP 协议。其实 UDP 无所谓什么协议，它没有自己的状态机，仅仅是在 IP 层上做了一些封装，不保证报文能确定到达，没有请求－应答机制，所有的行为，和 IP 应用协议一样。只不过，它多了一个 port 的概念，此 port 不是指主机上的网络端口，而是从操作系统内核的角度看到的应用程序“标识”。我们都知道如何调用操作系统的接口，但操作系统是如何“调用”应用程序的呢？在我们现在的 PC 机操作系统中，这是无法办到的。

于是人们为应用程序设置一个标识，内核根据这个标识确定是哪一个应用程序曾经给它发过请求，然后把数据发给应用程序，这样就避免操作系统把所有的数据发给所有在等待数据的应用程序，从操作系统的角度看，这个标识就是一个个的端口。比如你在网络上和一个女孩聊天，同时也和一个教授讨论问题。你当然不希望发给女孩的话教授也能收到，这就是传输层网络应用中加入的 port 概念。在 IP 层，每个应用的标识就是 ip 地址，内核根据 ip 来处理报文，要么给本机，要么转给别人，在 UDP 和 TCP 层，不仅要有 ip 地址，而且还要 port 号，内核根据 ip 地址确定了属于本机的报文后，还要根据 port 号确定哪一个应用程序才是报文的终极目的地。

5.1.1　UDP 用户代码

可能使用 U D P 的情况包括：转发路由表数据交换、系统信息、网络监控数据等的交换。这些类型的交换不需要流控、应答、重排序或任何 T C P 提供的功能。

我们先给出 UDP 应用程序的一般示例，如下面两个框图，左边是服务器端，右边是客户端。

服务器端典型代码

```
1.  socket(..., SOCK_DGRAM, 0)
2.  bind(..., &servaddr, ...)
3.  recvfrom(..., &clientaddr, ...);
```

客户端典型代码

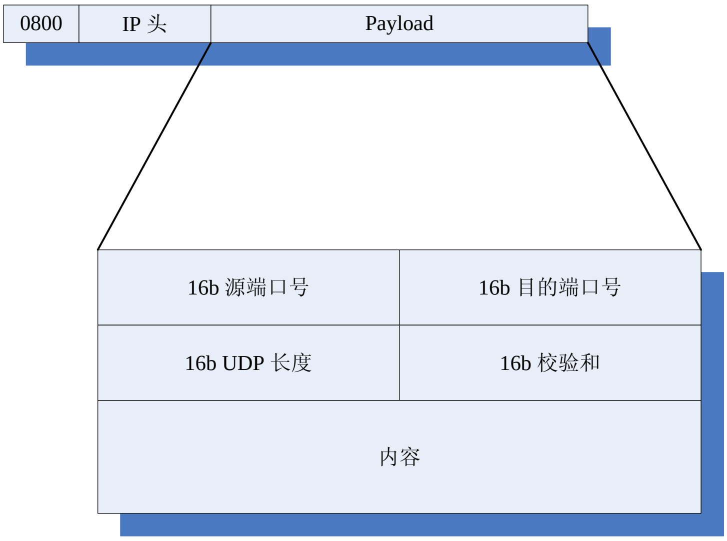
```
1.  socket(..., SOCK_DGRAM, 0);
2.  sendto(..., &servaddr, ...);
```

服务器端有 3 步，第一步和第 3 步我们都很熟悉，客户端更简单，都是我们曾经研究过的接口。出于篇幅，我就不详细说明这几个函数了，只是提醒大家要注意的接口参数，将导致内核中的 INET 层使用 inetsw_array[]数组里第二个单元——UDP 协议 inet_protosw{}结构，从而使用 udp_prot 去创建 socket 并用此 proto 指导报文的发送和接收。

唯一让我们有一丝兴趣的就是服务器端有一个 bind 接口，这是在基于 IP 层的应用程序中看不到的，它是否有什么特殊用途呢？

5.1.2 UDP 数据报文格式

UDP 提供的服务比之 IP 其实不多，也就是 UDP 报文头部增加传输层上多路分配或多路分离端口以及一个 CRC 校验值。U D P 首部的各字段如图 5 - 1 所示。



图表 5-1UDP 数据报文格式

UDP 端口号特殊的一个方面是其源端口号可以置为 0，表示没有指定端口，用于不期望响应且没有所需端口号的场合。一般人会认为，UDP 的端口号非常简单，简单到似乎没有必要存在，除非用于 socket 事务标识。UDP 和 IP 的区别就是这个端口号，我们下面要研究的内容也和这个端口号有密切的关系。

5.1.3 服务器端 bind 的实现

int bind(int sockfd,strUCt sockaddr *my_addr, int addrlen);　　Sockfd 是调用 socket 函数返回的 socket 描述符,my_addr 是一个指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针；addrlen 常被设置为 sizeof(struct sockaddr)。　　struct sockaddr 结构类型是用来保存 socket 信息的：

```
struct sockaddr {
    unsigned short sa_family; /* 地址族， AF_xxx */
    char sa_data[14]; /* 14 字节的协议地址 */ };
```

sa_family 一般为 AF_INET，代表 Internet（TCP/IP）地址族；sa_data 则包含该 socket 的 IP 地址和端口号。bind 是将一个 socket 的地址设为一个(addr,port)对，对一个已经绑定的 socket，不能再进行绑定. 再次绑定 bind 返回-1，error == EINVAL。另外还有一种结构类型：

```
struct sockaddr_in { short int sin_family; /* 地址族 */
        unsigned short int sin_port; /* 端口号 */
        struct in_addr sin_addr; /* IP 地址 */
        unsigned char sin_zero[8]; /* 填充 0 以保持与 struct sockaddr 同样大小 */
    };
```

这个结构更方便使用。sin_zero 用来将 sockaddr_in 结构填充到与 struct sockaddr 同样的长度，可以用 bzero()或 memset()函数将其置为零。指向 sockaddr_in 的指针和指向 sockaddr 的指针可以相互转换，这意味着如果一个函数所需参数类型是 sockaddr 时，你可以在函数调用的时候将

一个指向 sockaddr_in 的指针转换为指向 sockaddr 的指针；或者相反。

使用 bind 函数时，可以用下面的赋值实现自动获得本机 IP 地址和随机获取一个没有被占用的端口号：

my_addr.sin_port = 0; /* 系统随机选择一个未被使用的端口号 */

my_addr.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */

通过将 my_addr.sin_port 置为 0，函数会自动为你选择一个未占用的端口来使用。同样，通过将 my_addr.sin_addr.s_addr 置为 INADDR_ANY，系统会自动填入本机 IP 地址。

bind 系统调用在协议栈内核函数中的第一站是 inet_bind 函数，它首先要调用协议本身的 bind 函数(即 struct sock 的成员 sk_prot->bind)，但 UDP 和 TCP 协议本身不提供 bind 函数。接下来对 bind 系统调用传入的 IP 地址和端口作一个正确性和类型检查，需要注意的是，0-1023 号端口是只有超级用户才有权限执行绑定的。然后，令传入的 IP 地址赋给 inet_sock{}的成员 rcv_saddr(本地接收地址)和 saddr(本地发送地址)，接下来，要对端口号作一个处理，调用协议本身的 get_port 函数(即 sock{}的成员 sk_prot->get_port( )对传入的端口号作检查，如果能够使用，则绑定成功，如果不能使用，则会返回 EADDRINUSE 错误。

```
1.  /*
2.   *  Bind a name to a socket. Nothing much to do here since it's
3.   *  the protocol's responsibility to handle the local address.
4.   *
5.   *  We move the socket address to kernel space before we call
6.   *  the protocol layer (having also checked the address is ok).
7.   */
8.  
9.  asmlinkage long sys_bind(int fd, struct sockaddr __user *umyaddr, int addrlen)
10. {
11.     struct socket *sock;
12.     char address[MAX_SOCK_ADDR];
13.     int err;
    根据 fd 找到相应的 file{}，然后再找到对应的 socket{}结构
14.     if((sock = sockfd_lookup_light(fd, &err, ...))!=NULL)
15.     {
16.         if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0) {
    在此调用的是 inet_bind，对于 TCP 应用也是这样的
17.             err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
18.         }
19.
20.     }
21. }
```

代码段 5-1 sys_bind 函数

这里有意思的是代表 INET 域网络层套接字的结构体 inet_sock{}有两个端口号相关的成员 __u16 num 和__u16 sport。它们都代表套接字的本地端口号。num 是主机字节序，sport 是网络字节序。当套接字类型为 SOCK_RAW 时，它们代表的是协议号(icmp,igmp 等)，前面已经介绍过了。

```
1.  int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
2.  {
3.  struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
4.  struct sock *sk = sock->sk;
5.  struct inet_sock *inet = inet_sk(sk);
6.  unsigned short snum;
7.  int chk_addr_ret;
8.  int err;
9.  
10. /* If the socket has its own bind function then use it. (RAW) */
    只有 RAW 类型的 proto{}定义了 bind 函数指针，而 TCP 和 UDP 没有相应的 bind，这一点要记住
11. if (sk->sk_prot->bind) {
```

```
12.     err = sk->sk_prot->bind(sk, uaddr, addr_len);
13.     goto out;在 RAW IP 调用了 bind 后就退出了
14. }
15.
```

```
static int raw_bind(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
        struct inet_sock *inet = inet_sk(sk);
        struct sockaddr_in *addr = (struct sockaddr_in *) uaddr;
        inet->rcv_saddr=inet->saddr= addr->sin_addr.s_addr;
        if (chk_addr_ret == RTN_MULTICAST
               || chk_addr_ret == RTN_BROADCAST)
                inet->saddr = 0;    /* Use device */
        sk_dst_reset(sk);
        ret = 0;
        out:   return ret;
}
```

```
16. chk_addr_ret = inet_addr_type(addr->sin_addr.s_addr);
17.
18. /* Not specified by any standard per-se, however it breaks too
19.  * many applications when removed.  It is unfortunate since
20.  * allowing applications to make a non-local bind solves
21.  * several problems with systems using dynamic addressing.
22.  * (ie. your servers still start up even if your ISDN link
23.  *  is temporarily down)
24.  */
25. err = -EADDRNOTAVAIL;
26. if (!sysctl_ip_nonlocal_bind &&
27.     !inet->freebind &&
28.     addr->sin_addr.s_addr != INADDR_ANY &&
29.     chk_addr_ret != RTN_LOCAL &&
30.     chk_addr_ret != RTN_MULTICAST &&
31.     chk_addr_ret != RTN_BROADCAST)
32.     goto out;
   一般情况下 sin_port 的值不为 0，
33. snum = ntohs(addr->sin_port);
34. err = -EACCES;
    如果服务器的 port 大于 1024，那么应用程序必须有超级用户的权限
35. if (snum && snum < PROT_SOCK && !capable(CAP_NET_BIND_SERVICE))
36.     goto out;
37.
38. /*      We keep a pair of addresses. rcv_saddr is the one
39.  *      used by hash lookups, and saddr is used for transmit.
40.  *
41.  *      In the BSD API these are the same except where it
42.  *      would be illegal to use them (multicast/broadcast) in
43.  *      which case the sending device address is used.
44.  */
45.
46. /* 检查错误，比如 active socket, 重复 bind. */
47. err = -EINVAL;
    此时 sock 的状态依然为 CLOSE，而且 num 还是 0
48. if (sk->sk_state != TCP_CLOSE || inet->num)
49.     goto out_release_sock;
    这里和 raw_bind 函数中完成的工作一致
50. inet->rcv_saddr = inet->saddr = addr->sin_addr.s_addr;
51. if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
52.     inet->saddr = 0;  /* Use device */
53.
54. /* 到此时可以确定我们可以进行端口绑定了*/
   调用 udp_v4_get_port，见下面的解释，对于 UDP 和 TCP 来说，这个函数是最重要的
55. if (sk->sk_prot->get_port(sk, snum)) {
56.     ......
57. }
58.
59. if (inet->rcv_saddr)
60.     sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
61. if (snum)
62.     sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
```

```
    把从第 55 行得到的 inet→num 赋给 sport，而且目的端口和目的地址依然填 0
63. inet->sport = htons(inet->num);
64. inet->daddr = 0;
65. inet->dport = 0;
66. sk_dst_reset(sk);
67. err = 0;
68. out_release_sock:
69. release_sock(sk);
70. out:
71. return err;
72. }
```

代码段 5-2 inet_bind 函数

在下面的 udop_v4_get_port 函数中我们会遇到一个比较重要的数据结构 udp_hash 表，不过注意，在 2.6.18 的代码中是没有其初始化代码的，我也不知道是遗漏了还是我没有找到。这个全局变量的用处在于把 sk 保存起来，因为一个 sk 代表一个用户打开的 socket 接口，在接收报文的时候协议栈会查找此 hash 表，希望能把相应的报文传给正确的用户。

上一章中曾提到这个函数，所以大家可以结合起来看：

```
1.  static int udp_v4_get_port(struct sock *sk, unsigned short snum)
2.  {
3.  struct hlist_node *node;
4.  struct sock *sk2;
5.  struct inet_sock *inet = inet_sk(sk);
```

```
static int inet_autobind(struct sock *sk)
{
        struct inet_sock *inet = inet_sk(sk);
        if (!inet->num) {
                sk->sk_prot->get_port(sk, 0);
                inet->sport = htons(inet->num);
        }
        return 0;
}
```

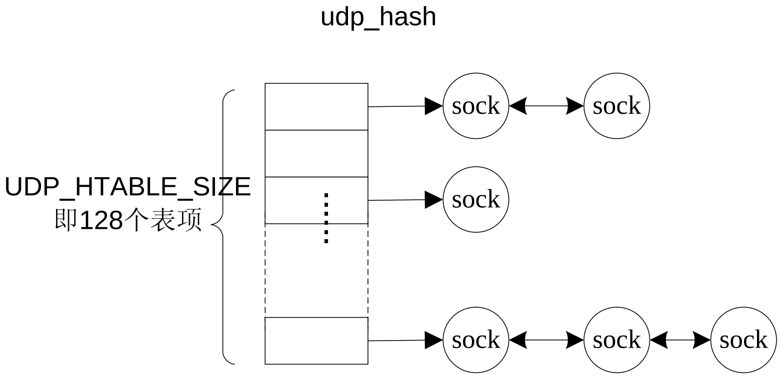
```
6.  if (snum == 0) {
        不管是 UDP 还是 TCP，它们的客户端一般都没有指定 port 号，所以它们一定要分配一个重来没有使
        用过的端口号。这个端口号是源端的端口号，不是目的端的端口号，目的端的端口号已经由用户层代
        码指定，请读者自行参考 udp_sendmsg 函数。那个地方是 Raw IP 和 TCP、UDP 最大最本质的区别。
7.      int best_size_so_far, best, result, i;
8.  
9.      if (udp_port_rover > sysctl_local_port_range[1] ||
10.         udp_port_rover < sysctl_local_port_range[0])
11.          udp_port_rover = sysctl_local_port_range[0];
12.     best_size_so_far = 32767;
13.     best = result = udp_port_rover;
        找到合适的 hash 下标
14.     for (i = 0; i < UDP_HTABLE_SIZE; i++, result++) {
15.         struct hlist_head *list;
16.         int size;
17.
18.         list = &udp_hash[result & (UDP_HTABLE_SIZE - 1)];
19.         if (hlist_empty(list)) {
20.             if (result > sysctl_local_port_range[1])
21.                 result = sysctl_local_port_range[0] +
22.                     ((result - sysctl_local_port_range[0])&(UDP_HTABLE_SIZE-1));
23.             goto gotit;
24.         }
25.         size = 0;
26.         sk_for_each(sk2, node, list)
27.             if (++size >= best_size_so_far)
28.                 goto next;
29.         best_size_so_far = size;
30.         best = result;
31.     next:;
32.     }
        找到一个节点作为未使用的 port 号
33.     result = best;
34.     for(i = 0; i < (1 << 16)/UDP_HTABLE_SIZE; i++, result += UDP_HTABLE_SIZE) {
```

```
#define UDP_HTABLE_SIZE            128
int sysctl_local_port_range[2] = { 1024, 4999 };
```

```
35.         if (result > sysctl_local_port_range[1])
36.             result = sysctl_local_port_range[0]
37.                 + ((result - sysctl_local_port_range[0])& (UDP_HTABLE_SIZE-1));
38.         if (!udp_lport_inuse(result))
39.             break;
40.     }
41.     if (i >= (1 << 16) / UDP_HTABLE_SIZE)
42.         goto fail;
43. gotit:
44.     udp_port_rover = snum = result;
45. } else {
         如果传入的 snum 不为 0，那么只要判断是否已经有对应 snum 的 sock 在使用，如果有，说明已经有
         sock 绑定了该 snum，就返回错误，否则该分支什么也不做。这也说明，能走到这个分支必然是
         inet_bind 调用的。
46.     sk_for_each(sk2, node,
47.             &udp_hash[snum & (UDP_HTABLE_SIZE - 1)]) {
48.     ......
49. }
    这行代码反映了此函数的真实含义：将 sin_port 值放入 inet→num 中。
50. inet->num = snum;
51. if (sk_unhashed(sk)) {
        如果还没有将此 sk 放到 hash 表中，现在就插入
52.     struct hlist_head *h = &udp_hash[snum & (UDP_HTABLE_SIZE - 1)];
53.
54.     sk_add_node(sk, h);
55.
56. }
57.     return 0;
58.
59. fail:
60.     return 1;
61. }
```

代码段 5-3 udp_v4_get_port

udp 协议提供 udp_v4_get_port 函数用于自动获取本地端口号。端口号有一个固定的数值范围，自动获取必须在这个范围内进行。数组 int sysctl_local_port_range[2]指定了本地端口号的范围。其默认值为 1024 到 4999。对于高可用性系统，它的值应该是 32768 到 61000(在 TCP 协议进行初始化时，会进行这项设置)。可通过修改文件/proc/sys/net/ipv4/ip_local_port_range 的内容来修改这个范围。



图表 5-2 udp_hash 数据结构

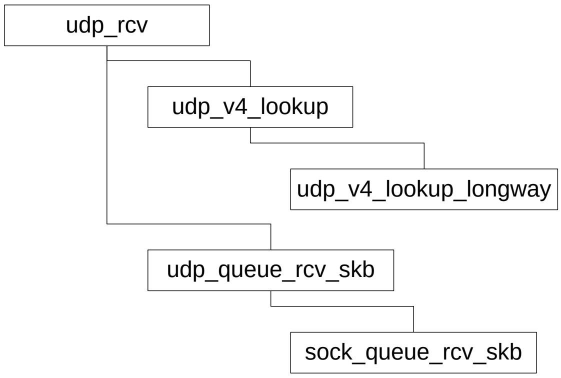
udp_hash 是一个 list 数组，总共有 128 项，所有在协议栈中建立的 udp socket 全部以本地端口号为关键字被放入这个哈希数组中，全局变量 udp_port_rover 记录了最近一次被分配的端口号。寻找一个新的可用的端口，总是从 udp_port_rover 开始找，检查 udp_hash[udp_port_rover & (UDP_HTABLE_SIZE - 1)]的 list 是否为空，如果为空，则取 udp_port_rover 为新的端口，如果不

为空，则记录下这个 list 的 size，同时保存下该端口号，然后遍历整个数组，找到 size 最小的一个 list，取对应的端口号为我们所要获得的端口。然后，检查这个新获得的端口号是否已经被使用（同样，通过检查 udp_hash 实现)。如果已在使用中，则把端口号加上 UDP_HTABLE_SIZE(128)再检查。直至获得未使用的端口号。此函数执行结果如上图。

在此要提醒读者的是 RAW IP 也有相关的 hash 函数和 hash 表（参见前面 inet_create 函数），分别是 raw_v4_hash 和 raw_v4_htable，RAW IP 的 hash 表大小为 MAX_INET_PROTOS（256），ping 的应用使用了 ICMP（1）的位置。

5.1.4 接收代码

从内核的方向来说，那么 udp_rcv 是头，在其中先查找对应的 sock{}，然后通过 udp_queue_rcv_skb 函数把报文放入进程的等待队列。



图表 5-3 udp_rcv 函数调用树

上文已经说过，udp_hash 表存放着用户打开的 sock{}结构，如果能在接收时查找到对应的 sock{}，那说明此报文有主，如果没有找到，那么它属于无人理会的报文，该丢弃就丢弃。

```
1.  static struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport,
2.                    u32 daddr, u16 dport, int dif)
3.  {
4.  struct sock *sk, *result = NULL;
5.  struct hlist_node *node;
6.  unsigned short hnum = ntohs(dport);
7.  int badness = -1;
    搜索整个 hash 表
8.  sk_for_each(sk, node, &udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]) {
9.      struct inet_sock *inet = inet_sk(sk);
10.
11.     if (inet->num == hnum) {
12.         int score = (sk->sk_family == PF_INET ? 1 : 0);
13.         if (inet->rcv_saddr) {
14.             if (inet->rcv_saddr != daddr)
15.                 continue;
16.             score+=2;
17.         }
18.         if (inet->daddr) {
19.             if (inet->daddr != saddr)
20.                 continue;
21.             score+=2;
22.         }
23.         if (inet->dport) {
24.             if (inet->dport != sport)
25.                 continue;
26.             score+=2;
27.         }
28.         if (sk->sk_bound_dev_if) {
29.             if (sk->sk_bound_dev_if != dif)
```

```
30.                 continue;
31.             score+=2;
32.         }
33.         if(score == 9) {
34.             result = sk;
35.             break;
36.         } else if(score > badness) {
37.             result = sk;
38.             badness = score;
39.         }
40.     }
41. }
42. return result;
43. }
```

代码段 5-4 udp_v4_lookup_longway 函数

上面是内核往用户方向的函数处理过程，下面是用户面往内核方向的函数处理流程。如何知道一个套接口的接收队列中有多少数据排队呢？ 有三种方法。

a. 如果在没有数据可读时还有其他事情可做，为了不阻塞在内核中，可以使用非阻塞 I/O。

b. 如果想检查一下数据而使数据流在队列中可以使用 MSG_PEEK 标志。

c. ioctl 中的 FIONREAD 命令。返回套接口接收队列中数据的字节数。之前大家可以套用之前 RAW IP 的 send 解析，找到 udp_recvmsg，它调用的是 skb_recv_datagram。这里要注意的是 inet_recvmsg 在 2.6.14 之后换成 sock_common_recvmsg，现在不过换了个函数名。

```
1.  /**
2.   *  skb_recv_datagram – 接收一个数据报 skbuff
3.   *  @sk: socket
4.   *  @flags: MSG_ flags
5.   *  @noblock: blocking operation?
6.   *  @err: error code returned
7.   *
8.   *  当一个 skb 返回以后该函数会锁住 socket，所以调用者需要解锁，通常调用 skb_free_datagram)
9.   *
10.  *  * 直到现在还没有锁住 socket，该函数没有处理抢占的情况，这种做法应该/可能在高负载情况下极
    大的提高数据报文 socket 的延迟，这是因为把内核空间的数据拷贝到用户空间会消耗大量时间。
11.  *  *                              --ANK (980729)
12.  *
13.  *  The order of the tests when we find no data waiting are specified
14.  *  quite explicitly by POSIX 1003.1g, don't change them without having
15.  *  the standard around please.
16.  */
17. struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags,
18.               int noblock, int *err)
19. {
20.    struct sk_buff *skb;
21.    long timeo;
22.
23.    timeo = sock_rcvtimeo(sk, noblock);
24.
25.    do {
26.     /* Again only user level code calls this function, so nothing
27.      * interrupt level will suddenly eat the receive_queue.
28.      *
29.      * Look at current nfs client by the way...
30.      * However, this function was corrent in any case. 8)
31.      */
32.        if (flags & MSG_PEEK) {
33.            unsigned long cpu_flags;
34.            skb = skb_peek(&sk->sk_receive_queue);
35.            if (skb)
36.                ......
```

```
37.             } else
38.              skb = skb_dequeue(&sk->sk_receive_queue);
39.
40.     if (skb)
41.         return skb;
42.      ......
43.     } while (!wait_for_packet(sk, err, &timeo));
44.
45.     return NULL;
46.
47. no_packet:
48.     return NULL;
49. }
```

代码段 5-5 skb_recv_datagram 函数

我们已经知道可以用__skb_pull 剥去 IP 首部（我们可以通过 skb->nh.raw 找到它）。最后，skb->h.raw 指向 data，即 udp 首部，udp 首部其实到最后都没有被剥去，应用程序在调用 recv 接收数据时，直接从 skb->data+sizeof(struc udphdr)的位置开始拷贝。

recv 和 send 函数提供了和 read 和 write 差不多的功能.不过它们提供了第四个参数来控制读写操作.

int recv(int sockfd,void *buf,int len,int flags)

int send(int sockfd,void *buf,int len,int flags)

前面的三个参数和 read,write 一样,第四个参数可以是 0 或者是以下的组合 ：

| MSG_DONTROUTE | 不查找路由表 ，是 send 函数使用的标志.这个标志告诉 IP 协议.目的主机在本地网络上面 ,没有必要查找路由表.这个标志一般用网络诊断和路由程序里面.

| MSG_OOB | 接受或者发送带外数据 |

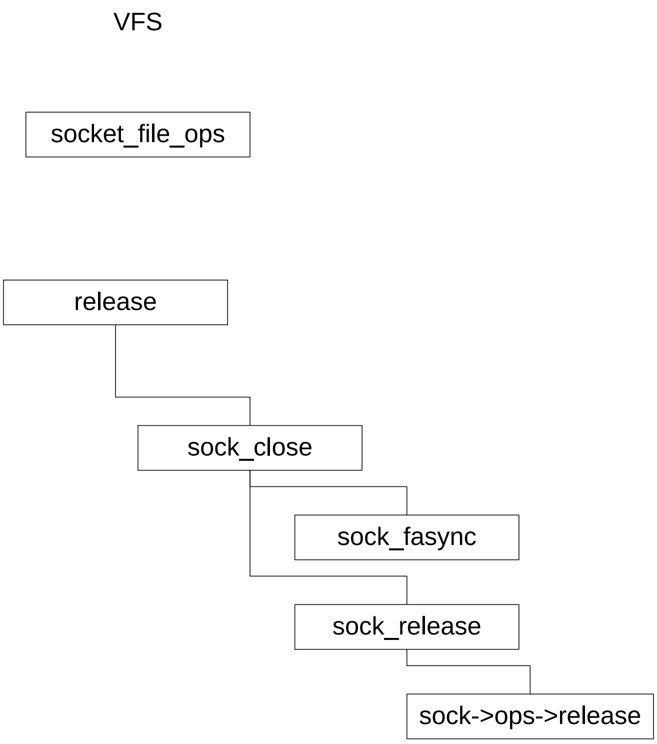
| MSG_PEEK | 查看数据,并不从系统缓冲区移走数据：是 recv 函数的使用标志,表示只是从系统缓冲区中读取内容,而不清楚系统缓冲

| MSG_WAITALL | 等待所有数据：是 recv 函数的使用标志,表示等到所有的信息到达时才返回.使用这个标志的时候 recv 回一直阻塞,直到指定的条件满足,或者是发生了错误：1)当读到了指定的字节时,函数正常返回.返回值等于 len； 2)当读到了文件的结尾时,函数正常返回.返回值小于 len； 3)当操作发生错误时,返回-1,且设置错误为相应的错误号(errno)

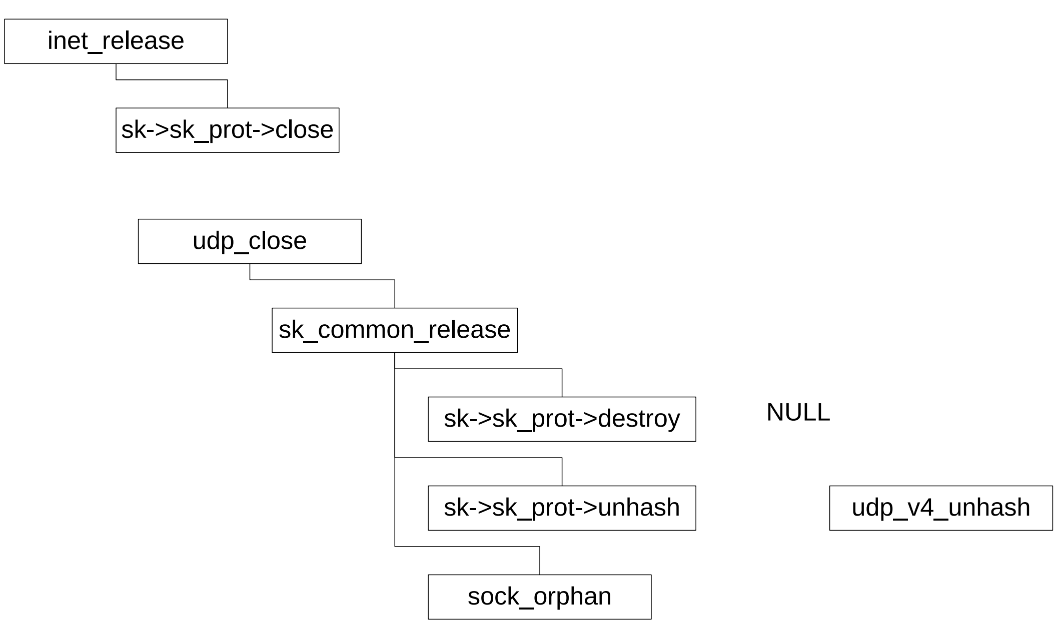
如果 flags 为 0,则和 read,write 一样的操作.还有其它的几个选项,不过我们实际上用的很少,可以查看 Linux Programmer's Manual 得到详细解释.

5.1.5 释放 UDP 的 socket

网络系统作为一个特殊的文件系统存在，不可避免的要和 VFS 打交道，那么在关闭 socket 时特别要关注 socket_file_ops 这个特别重要的全局变量。它是 VFS 和网络子系统的连接器，特别在应用层调用 close 这个系统调用的时候产生作用。



图表 5-4 release 函数



图表 5-5 inet_release 函数调用树

要注意的是关闭 raw IP 时使用的也是 sk_common_release，而且 SCTP 类型的 socket 也是调用这个函数去关闭。但是，TCP 不是这样，它相对复杂，我们会在下一节介绍。

前面曾经说过，udp_hash{}这个全局数据结构存放系统中使用的 port 号信息，这是在 udp_v4_get_port 中完成插入操作的。现在到了要把 port 释放的时候了，但是搜编整个代码，没有

看到其它 udp_hash 出现的地方，难道 Linux 涉及者容许 port 号只使用一次？非也非也，实际上的操作在 udp_v4_unhash 中，此函数调用 sk_del_node_init，其中又调用__sk_del_node_init，

```
1.  static __inline__ int __sk_del_node_init(struct sock *sk)
2.  {
     判断此节点是否真的挂在链表上
3.      if (sk_hashed(sk)) {
    就在此处从 udp_hash 中删掉自己，为将来能再次使用该 port 准备。
4.          __sk_del_node(sk);
5.          sk_node_init(&sk->sk_node);
6.          return 1;
7.      }
8.      return 0;
9.  }
```

__hlist_del(&sk->sk_node);

代码段 5-6 __sk_del_node_init 函数

虽然没有提到 udp_hash 这个全局变量，但是不要忘了 sk_node 是一个双向链表，那么聪明的读者该知道__sk_del_node 中做什么了吧？

5.2 更高阶的 TCP

TCP 和 UDP 是迥异的传输层协议，被设计为做不同的事情。二者的共性是都使用 I P 作为其网络层协议。TCP 和 UDP 之间的主要差别在于可靠性。 TCP 是高可用性的，而 UDP 是一个简单的、尽力数据报转发协议。这个基本的差别暗示 TCP 更复杂，需要大量功能开销，然而 UDP 是简单和高效的。建立一个 socket，如果没有用它来监听连入请求，那么就能用它来发连出请求。对于面向无连接的协议如 UDP 来说，这一 socket 操作并不做许多事，但对于面向连接的协议如 TCP 来说，这一操作包括了在两个应用间建立一个虚连接。

5.2.1 TCP 用户代码

TCP 跟 UDP 不一样的地方就是：TCP 是一个数据流，UDP 是数据报

TCP 的 recv 的字节数可以是多少就是多少，只要对方有发送，不是按 send 的次数来 recv 的，是按发送的字节数来 recv 的，也就是说你可以一方进行十次 send 调用，每次发送 5 个 byte，而接收方可以一次 recv 50 个 byte，也可以 50 次 recv，每次一个 byte。UDP 的 sendto 与 recvfrom 的次数是对应的，一次 sendto 就是一个数据报，这个数据包就只能 recvfrom 一次，你不要想着分几次来 recvfrom，那是不可能的，除非你自己实现的协议栈。即使用 MSG_PEEK，你也不要想着 recvfrom 部分数据，UDP 数据包是有边界的。 先看看一般的 TCP 应用程序是如何交互的，下面是两份伪代码，分别是服务器端和客户端的代码：

服务器端典型代码

```
1.  socket(..., SOCK_STREAM, 0)
2.  bind(..., &servaddr, ...)
3.  listen(...)
4.  accept(..., &clientaddr, ...);
5.  recv(..., &clientaddr, ...);
```

客户端典型代码

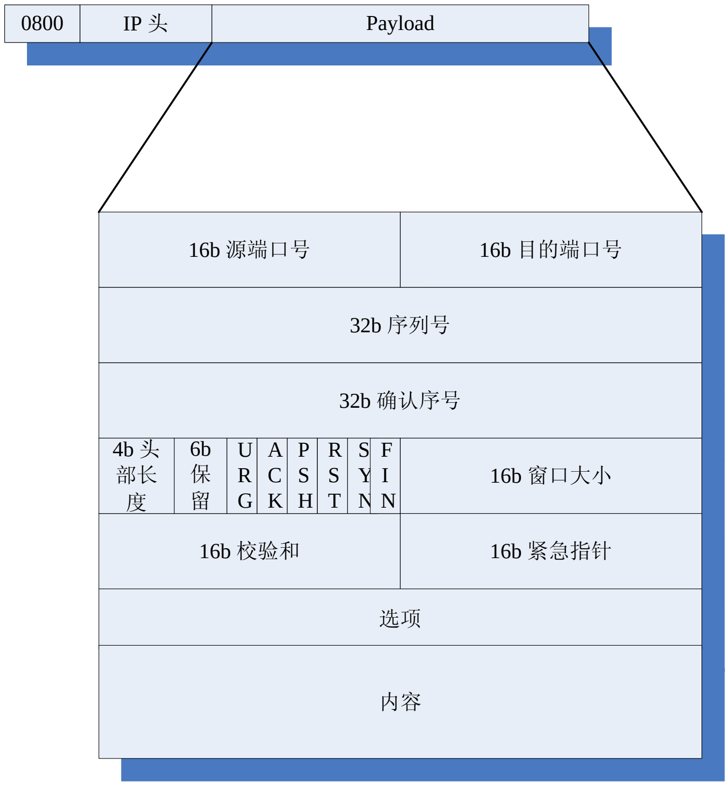
```
1.  socket(..., SOCK_STREAM, 0);
2.  connect( )
3.  send(..., &servaddr, ...);
```

和 UDP 程序的区别还是挺大的，首先是服务器端多了一个 listen 和 accept，客户端多了一个 connect。这几个函数底下完成什么工作呢？本章就尝试回答这个问题。

5.2.2 TCP 数据报文格式

传输控制协议(TCP)提供了可靠的报文流传输和对上层应用的连接服务，TCP 使用顺序的应答，能够按需重传报文。如果不计任选字段，它通常是 2 0 个字节

TCP 头如下所示：



图表 5-6TCP 数据报文格式

每个 TCP 段都包含源端和目的端的端口号，用于寻找发端和收端应用进程。这两个值加上 IP 首部中的源端 I P 地址和目的端 I P 地址唯一确定一个 TCP 连接。

有时，一个 IP 地址和一个端口号也称为一个插口（socket）。这个术语出现在最早的 TCP 规范（RFC 793）中，后来它也作为表示伯克利版的编程接口。插口对（socket pair）(包含客户 IP 地址、客户端口号、服务器 I P 地址和服务器端口号的四元组)可唯一确定互联网络中每个 TCP 连接的双方。

序号用来标识从 TCP 发端向 TCP 收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 TCP 用序号对每个字节进行计数。序号是 32 bit 的无符号数，序号到达 $2^{32}-1$ 后又从 0 开始。

当建立一个新的连接时， SYN 标志变 1。序号字段包含由这个主机选择的该连接的初始序号 ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为这个 ISN 加 1，因为 SYN 标志消耗了一个序号（将在下章详细介绍如何建立和终止连接，届时我们将看到 FIN 标志也要占用一个序号）。

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有 ACK 标志（下面介绍）为 1 时确认

序号字段才有效。

发送 ACK 无需任何代价，因为 32 bit 的确认序号字段和 ACK 标志一样，总是 TCP 首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置， ACK 标志也总是被设置为 1。

TCP 为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

TCP 可以表述为一个没有选择确认或否认的滑动窗口协议我们说 TCP 缺少选择确认是因为 TCP 首部中的确认序号表示发方已成功收。

到字节，但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。例如，如果 1～1024 字节已经成功收到，下一报文段中包含序号从 2049～3072 的字节，收端并不能确认这个新的报文段。它所能做的就是发回一个确认序号为 1025 的 ACK。它也无法对一个报文段进行否认。例如，如果收到包含 1025～2048 字节的报文段，但它的检验和错， TCP 接收端所能做的就是发回一个确认序号为 1025 的 ACK。

首部长度给出首部中 32 bit 字的数目。需要这个值是因为任选字段的长度是可变的。这个字段占 4 bit，因此 T C P 最多有 60 字节的首部。然而，没有任选字段，正常的长度是 2 0 字节。

在 TCP 首部中有 6 个标志比特。它们中的多个可同时被设置为 1。在随后的章节中有更详细的介绍。

TCP 的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端正期望接收的字节。窗口大小是一个 16 bit 字段，因而窗口大小最大为 6 5 5 3 5 字节。

检验和覆盖了整个的 TCP 报文段： TCP 首部和 TCP 数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。TCP 检验和的计算和 UDP 检验和的计算相似。

只有当 URG 标志置 1 时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。TCP 的紧急方式是发送端向另一端发送紧急数据的一种方式。

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。每个连接方通常都在通信的第一个报文段（为建立连接而设置 SYN 标志的那个段）中指明这个选项。它指明本端所能接收的最大长度的报文段。

从上图中我们注意到 TCP 报文段中的数据部分是可选的。我们将在下一节看到在一个连接建立和一个连接终止时，双方交换的报文段仅有 TCP 首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段

5.2.3 TCP 栈及 socket 的初始化

不知大家注意到没，UDP 协议没有特殊的初始化例程，而跟 TCP 有关的初始化似乎比较多，难道 UDP 是后娘养的？不管怎样，UDP 那一章没有初始化的内容，而 TCP 倒有，系统在启动时就有初始化 TCP 协议栈的动作，即在 inet_init 函数中通过两步实现，下面就是第一步——调用 tcp_v4_init：

```
1.  void __init tcp_v4_init(struct net_proto_family *ops)
2.  {
3.      这里创建的 socket 超出了本文的内容，大家可以跳过。
```

```
4.     int err = sock_create(PF_INET, SOCK_RAW, IPPROTO_TCP, &tcp_socket, 0);
5.  }
```

代码段 5-7 tcp_v4_init 函数

上面的函数没有完成什么实际的工作，那第二步呢？第二步是调用 tcp_init 函数：

```
1.  void __init tcp_init(void)
2.  {
3.     struct sk_buff *skb = NULL;
4.     unsigned long limit;
5.     int order, i, max_share;
6.
7.     if (sizeof(struct tcp_skb_cb) > sizeof(skb->cb))
8.         __skb_cb_too_small_for_tcp(sizeof(struct tcp_skb_cb),
9.                     sizeof(skb->cb));
   主要是申请 3 块内存，分别为 TCP 所需的 hash 表
10.    tcp_hashinfo.bind_bucket_cachep =
11.        kmem_cache_create("tcp_bind_bucket",
12.              sizeof(struct inet_bind_bucket), 0,
13.              SLAB_HWCACHE_ALIGN, NULL, NULL);
14.
15. /* Size and allocate the main established and bind bucket
16.  * hash tables.
17.  *
18.  * The methodology is similar to that of the buffer cache.
19.  */
20.    tcp_hashinfo.ehash =
21.        alloc_large_system_hash("TCP established",
22.                sizeof(struct inet_ehash_bucket),
23.                thash_entries,
24.                (num_physpages >= 128 * 1024) ?
25.                13 : 15,
26.                HASH_HIGHMEM,
27.                &tcp_hashinfo.ehash_size,
28.                NULL,
29.                0);
30.    tcp_hashinfo.ehash_size = (1 << tcp_hashinfo.ehash_size) >> 1;
31.    for (i = 0; i < (tcp_hashinfo.ehash_size << 1); i++) {
32.        INIT_HLIST_HEAD(&tcp_hashinfo.ehash[i].chain);
33.    }
34.
35.    tcp_hashinfo.bhash =
36.        alloc_large_system_hash("TCP bind",
37.                sizeof(struct inet_bind_hashbucket),
38.                tcp_hashinfo.ehash_size,
39.                (num_physpages >= 128 * 1024) ?
40.                13 : 15,
41.                HASH_HIGHMEM,
42.                &tcp_hashinfo.bhash_size,
43.                NULL,
44.                64 * 1024);
45.    tcp_hashinfo.bhash_size = 1 << tcp_hashinfo.bhash_size;
46.    for (i = 0; i < tcp_hashinfo.bhash_size; i++) {
47.        INIT_HLIST_HEAD(&tcp_hashinfo.bhash[i].chain);
48.    }
    以上的参数分别为：
       TCP estableished hash 表项为 65535 项，bind hash 表项为 32768 项
    注册拥塞控制处理函数，我们会在后面的章节中介绍
49.    tcp_register_congestion_control(&tcp_reno);
50. }
```

代码段 5-8 tcp_init 函数

而在系统部分完成 TCP 的初始化后，这些准备工作还不足以为客户提供一个完美的 TCP socket，所以当要真的使用 TCP 时，TCP 本身让用户在创建 socket 时有一个初始化 socket 的动作，

这个动作在用户调用 socket 系统函数时，借由其中的 inet_create 函数完成，请读者参考此函数的第 85 行（sk->sk_prot->init（)）。Tcp 的 init 函数指针指向 tcp_v4_init_sock，代码如下：

```
1.
2.  /* NOTE: A lot of things set to zero explicitly by call to
3.   *      sk_alloc() so need not be done here.
4.   */
5.  static int tcp_v4_init_sock(struct sock *sk)
6.  {
7.      struct inet_connection_sock *icsk = inet_csk(sk);
8.      struct tcp_sock *tp = tcp_sk(sk);
9.
10.     skb_queue_head_init(&tp->out_of_order_queue);
11.     tcp_init_xmit_timers(sk);
12.     tcp_prequeue_init(tp);
13.
14.     icsk->icsk_rto = TCP_TIMEOUT_INIT;
15.     tp->mdev = TCP_TIMEOUT_INIT;
16.
17. /* So many TCP implementations out there (incorrectly) count the
18.  * initial SYN frame in their delayed-ACK and congestion control
19.  * algorithms that we must have the following bandaid to talk
20.  * efficiently to them.  -DaveM
21.  */
22.     tp->snd_cwnd = 2;
23.
24. /* See draft-stevens-tcpca-spec-01 for
25.     discussion of the
26.  * initialization of these values.
27.  */
28.     tp->snd_ssthresh = 0x7fffffff;  /* Infinity
29.     tp->snd_cwnd_clamp = ~0;
30.     tp->mss_cache = 536;
31.
32.     tp->reordering = sysctl_tcp_reordering;
33.     icsk->icsk_ca_ops = &tcp_init_congestion_ops;
34.
35.     sk->sk_state = TCP_CLOSE;
36.
37.     sk->sk_write_space = sk_stream_write_spac
38.     sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
39.
40.     icsk->icsk_af_ops = &ipv4_specific;
41.
42.     return 0;
43. }
```

```
tcp_congestion_ops
tcp_init_congestion_ops  = {
.name        = "",
.ssthresh   = tcp_reno_ssthresh,
.cong_avoid = tcp_reno_cong_avoid,
.min_cwnd   = tcp_reno_min_cwnd,
};
```

```
struct inet_connection_sock_af_ops
ipv4_specific = {
.queue_xmit    = ip_queue_xmit,
.rebuild_header=inet_sk_rebuild_header,
.conn_request =tcp_v4_conn_request,
.syn_recv_sock=tcp_v4_syn_recv_sock,
.net_header_len=sizeof(iphdr),
.setsockopt    = ip_setsockopt,
.addr2sockaddr=inet_csk_addr2sockaddr,
.sockaddr_len=sizeof(sockaddr_in),
```

代码段 5-9 tcp_v4_init_sock 函数

这个初始化步骤主要指定关于 TCP 协议栈内部一些函数集合，这样可以根据系统实际情况配置 TCP 协议运行于哪一个下层协议之上。由于目前 TCP 主要在 IP 之上运行，那么其 icsk_af_ops 指向了 ipv4_specific 这个数据结构。如果那一天要跑在 IPv6 之上就把这个指针指向 IPv6 的相关的操作集合之上。

到这里，关于 TCP 我们应该具体看看 TCP 有关系统调用的内容了。

5.2.4 服务器端 bind 和 listen 的实现

listen 系统调用的声明如下：int listen(int sockfd, int queue_length);

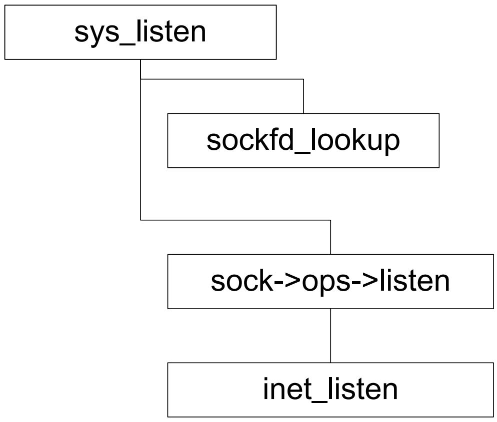
需要在此前调用 bind()函数将 sockfd 绑定到一个端口上，否则由系统指定一个随机的端口。

接收队列：一个新的 Client 的连接请求先被放在接收队列中，直到 Server 程序调用 accept 函数接受连接请求。

第二个参数 queue_length，指的就是接收队列的长度 也就是在 Server 程序调用 accept 函数之

前最大允许的连接请求数，多余的连接请求将被拒绝。

关于 bind 的 inet 实现已经在上一节 UDP 中介绍了，我们现在把 bind 和 listen 合在一节中介绍是有原因的。原因在于，TCP 内部的 bind 和 listen 关系比较密切。因为这两个系统调用都会操作一个全局数据结构：tcp_hashinfo。而且从内核代码中可以看出，只有 TCP 协议有 listen 接口，剩下 UDP 和 RAW IP 是不支持的，这样我们可以集中精力研究 listen 在 inet 层的实现。



图表 5-7 sys_listen 函数调用树

```
1.  /*
2.   *  把一个 socket 改变到侦听状态.
3.   */
4.  int inet_listen(struct socket *sock, int backlog)
5.  {
6.      struct sock *sk = sock->sk;
7.      unsigned char old_state;
8.      int err;
9.
    要保证此时 sock 的状态和类型，用户不能在客户端和服务器端连接后重复调用 listen
10.     if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)
11.         goto out;
    还要保证在 TCP 关闭或处于 LISTEN 状态才能继续
12.     old_state = sk->sk_state;
13.     if (!((1 << old_state) & (TCPF_CLOSE | TCPF_LISTEN)))
14.         goto out;
15.
16. /* 实际上，只有当 socket 处于 listen 状态时我们才能去调整 bakclog 的大小，而当只有处于 CLOSE
    状态才能调用第 19 行的函数
17.  */
18.     if (old_state != TCP_LISTEN) {
19.         err = inet_csk_listen_start (sk);
20.         .......
21.     }
    backlog 是由用户指定的
22.     sk->sk_max_ack_backlog = backlog;
23.     err = 0;
24.
25. out:
26.     release_sock(sk);
27.     return err;
28. }
```

代码段 5-10 inet_listen 函数

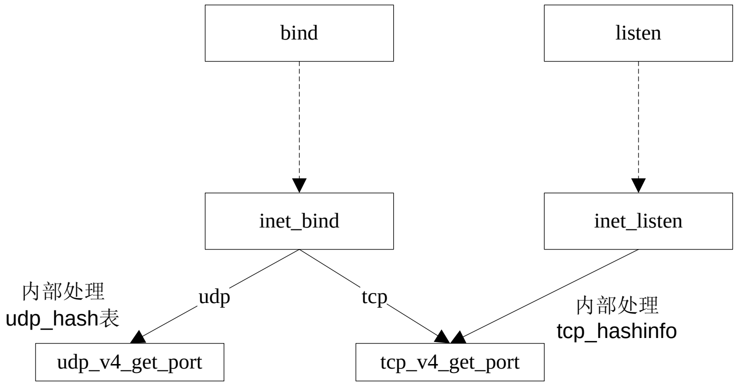
inet_csk_listen_start 的代码实现：

```
1.  int inet_csk_listen_start(struct sock *sk, const int nr_table_entries)
2.  {
3.      struct inet_sock *inet = inet_sk(sk);
```

```
4.     struct inet_connection_sock *icsk = inet_csk(sk);
5.     int rc = reqsk_queue_alloc(&icsk->icsk_accept_queue, nr_table_entries);
6.
7.     sk->sk_max_ack_backlog = 0;
8.     sk->sk_ack_backlog = 0;
9.     inet_csk_delack_init(sk);
10.
11. /* There is race window here: we announce ourselves listening,
12.  * but this transition is still not validated by get_port().
13.  * It is OK, because this socket enters to hash table only
14.  * after validation is complete.
15.  */
16.     sk->sk_state = TCP_LISTEN;
17.     if (!sk->sk_prot->get_port(sk, inet->num)) {
18.         inet->sport = htons(inet->num);
19.
20.         sk_dst_reset(sk);
        要么把自己加入到 tcp_hashinfo 中的 ehash 中，要么加入到 listening_hash 中，这要根据
        sk_state 的值来操作，如果是 LISTEN，就加入后者，如果是除 LISTEN 之外的值，那么就加入
        ehash 表，我们会在研究 connect 的代码中看到。
21.         sk->sk_prot->hash(sk);
22.
23.         return 0;
24.     }
25.
26.     sk->sk_state = TCP_CLOSE;
27.     __reqsk_queue_destroy(&icsk->icsk_accept_queue);
28.     return -EADDRINUSE;
29. }
```

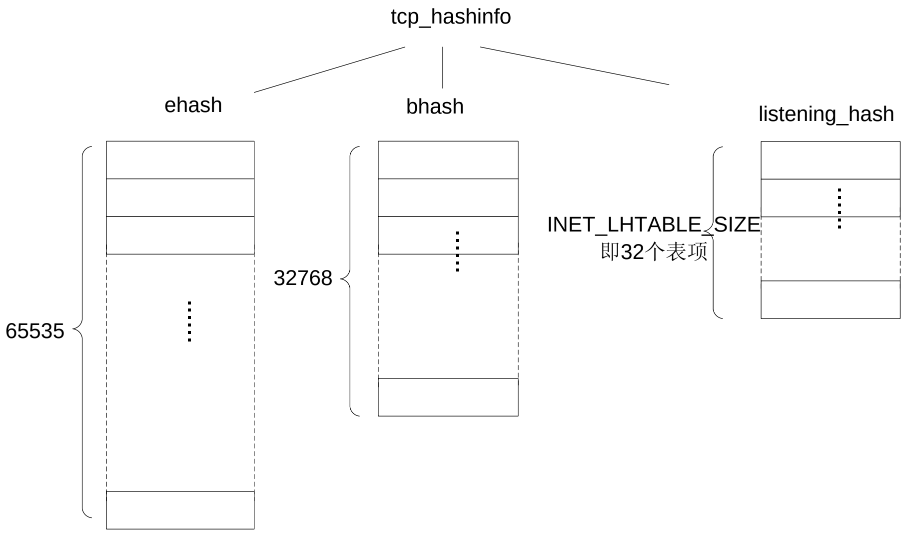
代码段 5-11 inet_csk_listen_start 函数

此时的 sk_prot→get_port 指的是 tcp_v4_get_port。它只是简单的调用了 inet_csk_get_port，并将 tcp_hashinfo 传入其中。下面这副图归纳了 UDP 和 TCP 下 bind 和 listen 的用处，它们会调用各自的 get_port 函数，但是注意的是 udp_prot→bind 和 tcp_proto→bind 这两个函数指针都是 NULL。



图表 5-8 bind 和 listen 都要调用 tcp_v4_get_port 函数

在 TCP 下，bind 和 listen 居然都调用了 get_port，这是怎么回事？让我们先了解一下 tcp_hashinfo 结构：



图表 5-9 tcp_hashinfo 中的内部数据结构

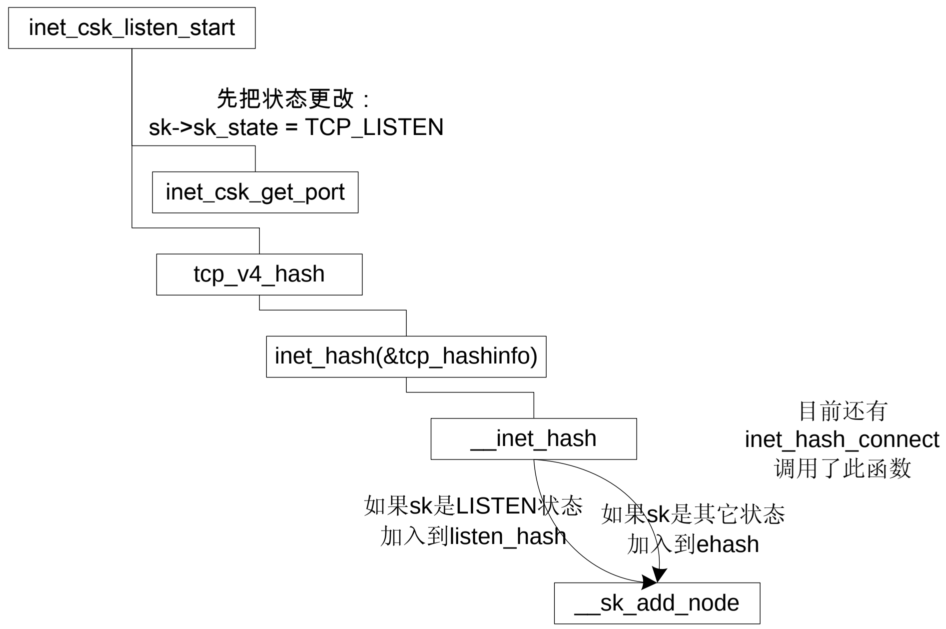
下面是 inet_csk_get_port 函数，传入它的 bind_conflict 参数是 inet_csk_bind_conflict，它有什么用呢？

```
1.
2.  /* Obtain a reference to a local port for the given sock,
3.   * 如果 snum 是 0，那么意味着可以选择任何有效的 port
4.   */
5.  int inet_csk_get_port(struct inet_hashinfo *hashinfo,
6.            struct sock *sk, unsigned short snum,
7.            int (*bind_conflict)(const struct sock *sk,
8.                     const struct inet_bind_bucket *tb))
9.  {
10.     struct inet_bind_hashbucket *head;
11.     struct hlist_node *node;
12.     struct inet_bind_bucket *tb;
13.     int ret;
14.
15.     if (!snum) {
        客户端的代码一般会进入此分支，因为客户端基本上都不知道将会得到什么样的端口，所以干脆填个
0       进来。请参考 UDP 的 get_port 函数
16.         int low = sysctl_local_port_range[0];
17.         int high = sysctl_local_port_range[1];
18.         int remaining = (high - low) + 1;
19.         int rover = net_random() % (high - low) + low;
20.
21.         do {
22.             head = &hashinfo->bhash[inet_bhashfn(rover, hashinfo->bhash_size)];
23.             inet_bind_bucket_for_each(tb, node, &head->chain)
24.                 if (tb->port == rover)
25.                     goto next;
26.             break;
27.     next:
28.             if (++rover > high)
29.                 rover = low;
30.         } while (--remaining > 0);
31.
32.     /* Exhausted local port range during search?  It is not
33.      * possible for us to be holding one of the bind hash
34.      * locks if this test triggers, because if 'remaining'
35.      * drops to zero, we broke out of the do/while loop at
36.      * the top level, not from the 'break;' statement.
37.      */
38.         ret = 1;
39.         if (remaining <= 0)
```

```
40.             goto fail;
41.
            我们得到了合法的 snum 值
42.         snum = rover;
43.     } else {
            服务器端代码进入此分支，但如果是第一次进入这个函数的话，下面的搜索不会找到任何结果
44.         head = &hashinfo->bhash[inet_bhashfn(snum, hashinfo->bhash_size)];
45.         inet_bind_bucket_for_each(tb, node, &head->chain)
46.             if (tb->port == snum)
47.                 goto tb_found;
48.     }
49.     tb = NULL;
        所以，第一次进入会跳转到 not found 标签
50.     goto tb_not_found;
51. tb_found:
52.     if (!hlist_empty(&tb->owners)) {
53.         if (sk->sk_reuse > 1)
54.             goto success;
55.         if (tb->fastreuse > 0 &&
56.             sk->sk_reuse && sk->sk_state != TCP_LISTEN) {
57.             goto success;
58.         } else {
            Listen 系统调用进入这个分支，它执行了 inet_csk_bind_conflict 函数。
59.             ret = 1;
60.             if (bind_conflict(sk, tb))
61.                 goto fail_unlock;
62.         }
63.     }
64. tb_not_found:
65.     ret = 1;
        在这里创建对应 port 的 hash 表项
66.     if (!tb && (tb = inet_bind_bucket_create
67.                 (hashinfo->bind_bucket_cachep, head, snum)) == NULL)
68.         goto fail_unlock;
69.     if (hlist_empty(&tb->owners)) {
70.         if (sk->sk_reuse && sk->sk_state != TCP_LISTEN)
71.             tb->fastreuse = 1;
72.         else
73.             tb->fastreuse = 0;
74.     } else if (tb->fastreuse &&
75.            (!sk->sk_reuse || sk->sk_state == TCP_LISTEN))
76.         tb->fastreuse = 0;
77. success:
        把自己加入到 tcp_hashinfo 中的 bhash 表
78.     if (!inet_csk(sk)->icsk_bind_hash)
79.         inet_bind_hash(sk, tb, snum);
80.
81.     ret = 0;
82. fail:
83.     return ret;
84. }
```

代码段 5-12 inet_csk_get_port 函数

调用 TCP bind 接口的结果如上图，先通过 inet_bind_hash 函数把 sock{}放入 tcp_hashinfo→bhash，在 sk_prot→get_port 结束之后调用 sk->sk_prot->hash，它的目的是往 tcp_hashinfo→listen_hash 中加入自己。在这里 sk_prot->hash 是 tcp_v4_hash。



图表 5-10 inet_csk_listen_start 函数调用树

tcp 的 listen 系统调用会再次调用 inet_csk_get_port 函数，这时候，会在 bhash 表中发现这个端口，此时的逻辑是虽然这个端口的 fastreuse 等于 1，并且 socket 的 sk_reuse 也等于 1，但 socket 此时已处于 TCP_LISTEN 状态。处于 TCP_LISTEN 状态的 socket 是不能共用别人的端口号的，所以，需要调用 inet_csk_bind_conflict 进行冲突处理。冲突处理的逻辑是从 inet_bind_bucket{}结构的 owners 中遍历绑定在该端口上的 socket，如果某 socket 跟当前的 socket 不是同一个，并且是绑定在同一个网络设备接口上的，并且它们两个之中至少有一个的 sk_reuse 表示自己的端口不能被重用，或者绑定在 onwers 上的那个 socket 已经是 TCP_LISTEN 状态了，并且它们两个之中至少有一个没有指定接收 IP 地址，或者两个都指定接收地址，但是接收地址是相同的，则冲突产生，否则不冲突。

也就是说，不使用同一个接收地址的 socket 可以共用端口号，绑定在不同的网络设备接口上的 socket 可以共用端口号。或者两个 socket 都表示自己可以被重用，并且还不在 TCP_LISTEN 状态，则可以重用端口号。

tcp 的 listen 系统调用到达冲突处理这一步时，owners 上的 socket 跟它明显是同一个，所以不冲突。继续执行，它又重新把 fastreuse 置 0，成功返回，所以 listen 系统调用的这一步是确保 socket 独占这个端口号。这是合理的，因为重用也只是在 socket 进入 TCP_TIME_WAIT 之后，它所占用的端口号能被其它 socket 重用。

5.2.5 服务器端 accept 的实现

accept 的函数声明如下：int accept(int sockfd,struct sockaddr *addr,int *addrlen);

此函数将响应连接请求，建立连接并产生一个新的 socket 描述符来描述该连接，该连接用来与特定的 Client 交换信息。

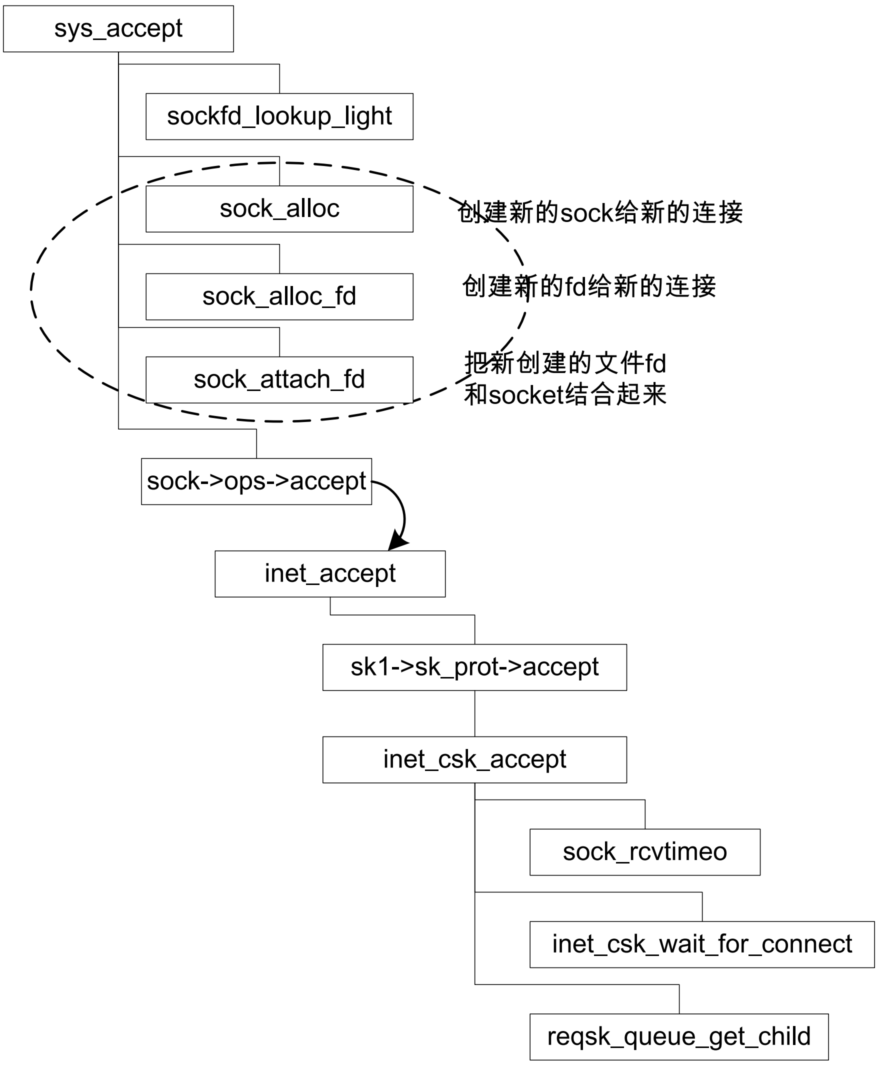
函数返回新的连接的 socket 描述符，错误返回-1

addr 将在函数调用后被填入连接对方的地址信息，如对方的 IP、端口等。

addrlen 作为参数表示 addr 内存区的大小，在函数返回后将被填入返回的 addr 结构的大小。

accept 缺省是阻塞函数，阻塞直到有连接请求。

内核中创建 TCP 的 socket 时，会在 inetsw_array 中找到相应的 proto_ops，即 inet_stream_ops，那么其 accept 就指向了 inet_accept。还要记住，UDP 和 RAW IP 的 socket ops 虽然指定了其 accpet 成员函数指针，但都是指向了 sock_no_accept，里面只有一句话：return -EOPNOTSUPP;。这种方式和 listen 的实现一样。



图表 5-11 sys_accept 函数调用树

```
1.  /*
2.   *  等待收到一个连接请求，要避免竞争条件，调用之前必须要锁住 socket
3.   */
4.  static int inet_csk_wait_for_connect(struct sock *sk, long timeo)
5.  {
6.      struct tcp_opt *tp = tcp_sk(sk);
7.      DEFINE_WAIT(wait);
8.      int err;
9.
10.     /*
11.      * 这是真正的“唤醒一个等待者”的机制：只有一个进程被唤醒，不是所有阻塞在此的进程。
12.      * Since we do not 'race & poll' for established sockets
13.      * anymore, the common case will execute the loop only once.
14.      *
15.      * Subtle issue: "add_wait_queue_exclusive()" will be added
16.      * after any current non-exclusive waiters, and we know that
```

```
17.      * it will always _stay_ after any new non-exclusive waiters
18.      * because all non-exclusive waiters are added at the
19.      * beginning of the wait-queue. As such, it's ok to "drop"
20.      * our exclusiveness temporarily when we get woken up without
21.      * having to remove and re-insert us on the wait queue.
22.      */
23.     for (;;) {
24.         把自己加入到等待队列，并且设置自己的状态是可中断的
25.         prepare_to_wait_exclusive(sk->sk_sleep, &wait,
26.                           TASK_INTERRUPTIBLE);
27.         release_sock(sk);
28.         if (!tp->accept_queue)
29.             timeo = schedule_timeout(timeo);
30.         lock_sock(sk);
31.         err = 0;
32.         if (tp->accept_queue)
33.             break;
34.         err = -EINVAL;
35.         if (sk->sk_state != TCP_LISTEN)
36.             break;
37.         err = sock_intr_errno(timeo);
38.         if (signal_pending(current))
39.             break;
40.         err = -EAGAIN;
41.         if (!timeo)
42.             break;
43.     }
        下面把任务设置成 TASK_RUNNING 状态，然后把当前 sock 从等待队列中删除
44.     finish_wait(sk->sk_sleep, &wait);
45.     return err;
46. }
```

```
fastcall signed long __sched schedule_timeout(signed long timeout)
{
	struct timer_list timer;
	unsigned long expire;

	switch (timeout)
	{
	case MAX_SCHEDULE_TIMEOUT:
		schedule();
		goto out;
	default:
	}
	 ......
	return timeout < 0 ? 0 : timeout;
}
```

代码段 5-13 inet_csk_wait_for_connect 函数

用户发起的 accept 操作就停在上面 schedule_timeout 中，其实现代码在右边框中，由于我们一般没有设置 timeout 值，所以是 MAX_SCHEDULE_TIMEOUT 的情况，这表示立即进入重新调度，而当前的进程可以处于睡眠，直到被其它事件唤醒。

5.2.6 客户端 connect 的实现——发起三次握手

connect 的函数声明如下：int connect(int sockfd, struct sockaddr *serv_addr,int addrlen);

sockfd 就不用再说了；serv_addr 是包含远端主机 IP 地址和端口号的指针；addrlen 是远端地质结构的长度。connect 函数在出现错误时返回-1，并且设置 errno 为相应的错误码。进行客户端程序设计无须调用 bind()，因为这种情况下只需知道目的机器的 IP 地址，而客户通过哪个端口与服务器建立连接并不需要关心，socket 执行体为你的程序自动选择一个未被占用的端口，并通知你的程序数据什么时候到达端口。

在探讨三次握手之前，我们先澄清一个概念：UDP 能调用 connect 吗？答案是：可以！但这有什么意义吗？sys_connect 会调用 inet_connect，当发现是 UDP 协议调用的，它会调用相关的 UDP 函数。最终调用了 ip_route_connect，它内部就是去查找路由 cache 表，如果找不到，就查找 FIB 表，如果找到就放入路由 cache 中。这不正是我们在讨论 raw_sendmsg 时已经介绍过的内容吗？对了，此函数确实也没什么新意，无非是查路由表嘛。

一个连出连接操作只能由一个在正确状态下的 INET BSD socket 来完成；换句话说，socket 不能是已建立连接的，并且有被用来监听连入连接。这意味着 BSD socket 结构必须是 SS_UNCONNECTED 状态。UDP 协议没有在两个应用间建立虚连接，任何发出的消息都是数据报，这些消息可能到达也可能不到达目的地。但它不支持 BSD socket 的 connect 操作。建立

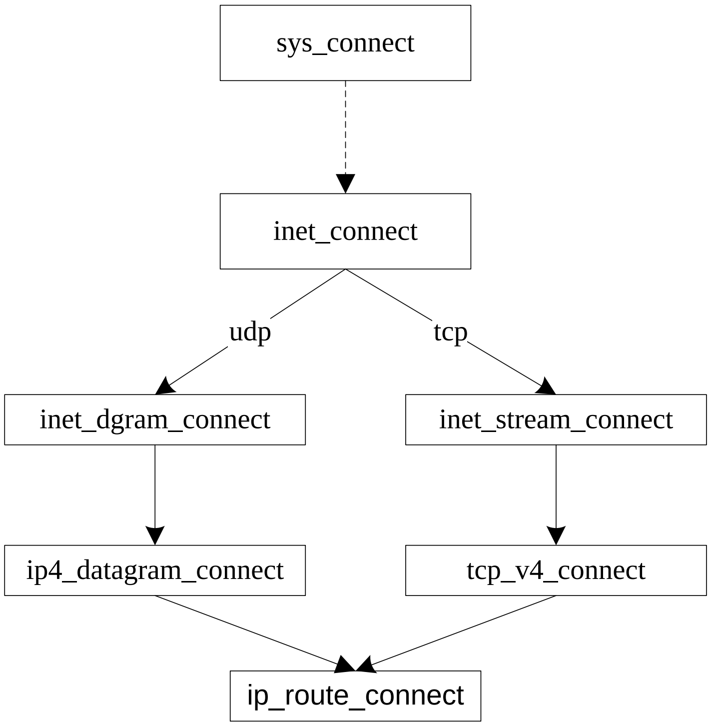
在 UDP 的 INET BSD socket 上的连接操作简单地设置远程应用的地址：IP 地址和 IP 端口号。另外，它还设置路由表入口的 cache 以便这一 BSD socket 在发送 UDP 包时不用再次查询路由数据库（除非这一路由已经无效）。INET sock 结构中的 ip_route_cache 指针指向路由缓存信息。如果没有给出地址信息，缓存的路由和 IP 地址信息将自动地被用来发送消息。UDP 将 sock 的状态改为 TCP_ESTABLISHED。

对于基于 TCP BSD socket 的连接操作，TCP 必须建立一个包括连接信息的 TCP 消息，并将它送到目的 IP。TCP 消息包含与连接有关的信息，一个唯一标识的消息开始顺序号，通过初始化主机来管理的消息大小的最大值，及发送与接收窗口大小等等。在 TCP 内，所有的消息都是编号的，初始的顺序号被用来作为第一消息号。Linux 选用一个合理的随机值来避免恶意协议冲突。每一从 TCP 连接的一端成功地传到另一端的消息要确认其已经正确到达。未确认的消息将被重传。发送与接收窗口的大小是第一个确认到达之前消息的个数。消息尺寸的最大值与网络设备有关，它们在初始化请求的最后时刻确定下来。如果接收端的网络设备的消息尺寸最大值更小，则连接将以小的一端为准。应用程序发出连接请求后必须等待目标应用程序的接受或拒绝连接的响应。TCP sock 期望着一个输入消息，它被加入 tcp_listening_hash 以便输入 TCP 消息能被指向这一 sock 结构。TCP 同时也开始计时，当目标应用没有响应请求，则连出连接请求超时。

socket 与地址绑定后，能监听指定地址的连入连接请求。一个网络应用程序能监听 socket 而不用先将地址 与之绑定；在这个例子中，INET socket 层找到一个未用的端口号（对这一协议）并自动将它与 socket 绑定。 监听 socket 函数将 socket 状态设成 TCP_LISTEN ，并做其它连入连接所需要的工作。

对于 UDP sockets，改变 socket 的状态就足够了，而 TCP 现在加了 socket 的 sock 数据结构到两个 hash 表中并激活，tcp_bound_hash 表和 tcp_listening_hash 表。这两个表都通过一个基于 IP 端口号的 hash 函数来索引。

无论何时，一个激活的监听 socket 接收一个连入的 TCP 连接请求，TCP 都要建立一个新的 sock 结构来描述它。最终接收时，这个 sock 结构将成为 TCP 连接的底层。它也复制包含连接请求的 sk_buff ，并将它放到监听 sock 结构的 receive_queue 中排队。复制的 sk_buff 包含一个指向新建立的 sock 结构的指针。



图表 5-12 sys_connct 在不同的协议下的执行路径

不过 UDP 调用了 connect 后，对 UDP 的性能有一些提高，证明在 udp_sendmsg 函数中：

```
1.  int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
2.      size_t len)
3.  {
4.      int connected = 0;
5.       ......
6.      {
7.          if (sk->sk_state != TCP_ESTABLISHED)
8.              return -EDESTADDRREQ;
9.          daddr = inet->daddr;
10.         dport = inet->dport;
11.         /* Open fast path for connected socket.
12.            Route will not be used, if at least one option is set.
13.          */
14.         connected = 1;
15.      }
16.     ......
17.     if (connected)
18.         rt = (struct rtable*)sk_dst_check(sk, 0);
        如果曾经进行过connect,那么rt肯定不是NULL(除非没有路由,但如果没有路由你还发送什么呢？）
19.     if (rt == NULL) {
        以下行为和在 raw_sendmsg 函数中一模一样
20.         struct flowi fl = { ..... };
21.         err = ip_route_output_flow(&rt, &fl, sk, !(msg->msg_flags&MSG_DONTWAIT));
22.
23.         if (connected)
24.             sk_dst_set(sk, dst_clone(&rt->u.dst));
25.     }
26.     err = udp_push_pending_frames(sk, up);
27.     ......
28. }
```

代码段 5-14 udp_sendmsg 函数部分代码

当发现曾经 connect 过，那么查路由表的步骤就省略了，那么发送性能就提高了。

在发送报文到一个已打开 socket 之前，一个使用 SOCK_STREAM 类型的 socket 的应用程序需要和对端连接。当连接建立的时候，要做的一件事是创建到目的地址的连接，而且，使用 SOCK_DGRAM 的应用程序在发送数据的时候，会使用 connect 函数建立到目的地的路由。下面看 TCP 的 connect 的内部实现。

```
1.  /*
2.   *  Connect to a remote host. There is regrettably still a little
3.   *  TCP 'magic' in here.
4.   */
5.  int inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,
6.          int addr_len, int flags)
7.  {
8.      struct sock *sk = sock->sk;
9.      int err;
10.     long timeo;
11.
12.     switch (sock->state) {
13.     default:
14.         err = -EINVAL;
15.         goto out;
16.     case SS_CONNECTED:
17.         err = -EISCONN;
18.         goto out;
19.     case SS_CONNECTING:
20.         err = -EALREADY;
21.         /* Fall out of switch with err, set for this state */
22.         break;
23.     case SS_UNCONNECTED:
            当前 sock 只能是这个状态，否则会出错
24.         err = -EISCONN;
25.         if (sk->sk_state != TCP_CLOSE)
                    goto out;
          调用下面要介绍的 tcp_v4_connect 函数。
26.         err = sk->sk_prot->connect(sk, uaddr, addr_len);
27.         ......
            注意这里的连接是 ing 状态
28.         sock->state = SS_CONNECTING;
29.
            只是进入 SS_CONNECTING 状态，和 SS_CONNECTED 唯一的区别是在非阻塞的情况下返回值是
            EINPROGRESS，而不是 EALREADY
30.         err = -EINPROGRESS;
31.         break;
32.     }
33.     if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)) {
34.         if (!timeo || !inet_wait_for_connect(sk, timeo))
35.             goto out;
36.     }
37.
38.     /* 连接被 RST，超时，ICMP 错误或者另外的进程关闭，
39.      */
40.     if (sk->sk_state == TCP_CLOSE)
41.         goto sock_error;
        到这里连接就是 ed 状态了
42.     sock->state = SS_CONNECTED;
43.     err = 0;
44. out:
45.     return err;
46.
47. sock_error:
48.     goto out;
49. }
```

代码段 5-15 inet_stream_connect 函数

UDP 的 connect 的内容很少，只是查询内核路由表，而对于 TCP 应用来说，connect 要完成的

任务非常多，调用 ip_route_connect 查询路由表是第一步。

```
1.  /* 此函数会发起一个连接. */
2.  int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
3.  {
4.      struct inet_sock *inet = inet_sk(sk);
5.      struct tcp_sock *tp = tcp_sk(sk);
6.      struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
7.      struct rtable *rt;
8.      u32 daddr, nexthop;
9.      int tmp;
10.     int err;
11.
12.     if (addr_len < sizeof(struct sockaddr_in))
13.         return -EINVAL;
14.
15.     nexthop = daddr = usin->sin_addr.s_addr;
16.     if (inet->opt && inet->opt->srr) {
17.         if (!daddr)
18.             return -EINVAL;
19.         nexthop = inet->opt->faddr;
20.     }
        查询路由表
21.     tmp = ip_route_connect(&rt, nexthop, inet->saddr,
22.                   RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
23.                   IPPROTO_TCP,
24.                   inet->sport, usin->sin_port, sk);
25.     if (tmp < 0)
26.         return tmp;
        不能是组播路由或广播路由
27.     if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
28.         return -ENETUNREACH;
29.     }
30.
31.     if (!inet->opt || !inet->opt->srr)
32.         daddr = rt->rt_dst;
        如果报文没有带源地址，就用路由表项里的源地址
33.     if (!inet->saddr)
34.         inet->saddr = rt->rt_src;
35.     inet->rcv_saddr = inet->saddr;
36.
37.     if (tp->rx_opt.ts_recent_stamp && inet->daddr != daddr) {
38.         /* Reset inherited state */
39.         tp->rx_opt.ts_recent      = 0;
40.         tp->rx_opt.ts_recent_stamp = 0;
41.         tp->write_seq         = 0;
42.     }
43.
44.     if (tcp_death_row.sysctl_tw_recycle &&
45.        !tp->rx_opt.ts_recent_stamp && rt->rt_dst == daddr) {
46.         struct inet_peer *peer = rt_get_peer(rt);
47.
48.         /* VJ's idea. We save last timestamp seen from
49.          * the destination in peer table, when entering state TIME-WAIT
50.          * and initialize rx_opt.ts_recent from it, when trying new connection.
51.          */
52.
53.         if (peer && peer->tcp_ts_stamp + TCP_PAWS_MSL >= xtime.tv_sec) {
54.             tp->rx_opt.ts_recent_stamp = peer->tcp_ts_stamp;
55.             tp->rx_opt.ts_recent = peer->tcp_ts;
56.         }
57.     }
        这里指定目的端口和地址
58.     inet->dport = usin->sin_port;
59.     inet->daddr = daddr;
60.
61.     inet_csk(sk)->icsk_ext_hdr_len = 0;
62.     if (inet->opt)
63.         inet_csk(sk)->icsk_ext_hdr_len = inet->opt->optlen;
```

```
64.
65.     tp->rx_opt.mss_clamp = 536;
66.
67.     /* 到此 socket 的二元组还没有唯一确定，因为 sport 可能是 0，但是我们会设置 socket 的状态
    为 SYN-SENT，并且不会
68.      * However we set state to SYN-SENT and not releasing socket
69.      * lock select source port, enter ourselves into the hash tables and
70.      * complete initialization after this.
71.      */
72.     tcp_set_state(sk, TCP_SYN_SENT);
        tcp_death_row 是一个全局变量，它有一个指向 tcp_hashinfo 的指针，这个函数会用到它
73.     err = inet_hash_connect(&tcp_death_row, sk);
74.
75.     err = ip_route_newports(&rt, IPPROTO_TCP, inet->sport, inet->dport, sk);
76.
77.     /* OK, now commit destination to socket.  */
78.     sk->sk_gso_type = SKB_GSO_TCPV4;
79.     sk_setup_caps(sk, &rt->u.dst);
        生成一个序列号，用作将来的 3 次握手协议
80.     if (!tp->write_seq)
81.         tp->write_seq = secure_tcp_sequence_number(inet->saddr,
82.                               inet->daddr,
83.                               inet->sport,
84.                               usin->sin_port);
85.
86.     inet->id = tp->write_seq ^ jiffies;
87.
88.     err = tcp_connect(sk);
89.     rt = NULL;
90.
91.     return 0;
92. failure:
93.
     把这个 sock 从 hash 表中剔除，
    并且释放本地端口
94. tcp_set_state(sk, TCP_CLOSE);
95. ......
96. return err;
97. }
```

```
int tcp_connect(struct sock *sk)
{
    ......
    tcp_transmit_skb(sk, buff, 1, GFP_KERNEL);
    发起一个定时器，重复发送 SYN，直到收到一个应答
    inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
                              inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
}
```

代码段 5-16 tcp_v4_connect 函数

sysctl_local_port_range 是从 32768～61000。

IP 为 TCP 和 UDP 提供 ip_route_connect 去建立一条路由。实际上，此函数只是主要路由解析函数 ip_route_output_flow 和 ip_route_output_key 的封装函数，实际上就是“可能”调用 2 次 ip_route_output_key 函数，第一次调用会找到下一跳的 ip 地址，也许在路由缓存或 fib 中找到，然后第二次找到下一跳的具体邻居，到 neigh_table 中找到：

```
1.  static inline int ip_route_connect(struct rtable **rp, u32 dst,
2.                u32 src, u32 tos, int oif, u8 protocol,
3.                u16 sport, u16 dport, struct sock *sk)
4.  {
5.      struct flowi fl = { .oif = oif,
6.              .nl_u = { .ip4_u = { .daddr = dst,
7.                       .saddr = src,
8.                       .tos   = tos } },
9.              .proto = protocol,
10.             .uli_u = { .ports =
11.                   { .sport = sport,
12.                  .dport = dport } } };
13.
14.     int err;
15.     if (!dst || !src) {
        如果源地址或目的地址都是 0，那么我们必须先找出相应的目的和源，把 flowi 填好以后再去找
        邻居地址。当 dst 为 0，那么目的和源都是环回地址，请参考 ip_route_output_slow
16.         err = __ip_route_output_key(rp, &fl);
17.
18.         fl.fl4_dst = (*rp)->rt_dst;
19.         fl.fl4_src = (*rp)->rt_src;
20.         ip_rt_put(*rp);
21.         *rp = NULL;
22.     }
23.     return ip_route_output_flow(rp, &fl, sk, 0);
24. }
```

这两个函数在 RAW IP 发送过程中都曾见过，不要忘记了。

当我们还没有发送建立 connect 的请求，我们就先把 sock{}的状态置为 TCP_SYN_SENT，然后把此 sock{}放入上一节中介绍的 tcp_hashinfo 中的 ehash 表中。不过这一切没有这么直观，因为 tcp_hashinfo 是放在 tcp_death_row 这个全局结构中的。

```
1.  /*
2.   * 把一个端口绑定到一个 sock 上，然后放入 hash 表中
3.   */
4.  int inet_hash_connect(struct inet_timewait_death_row *death_row,
5.          struct sock *sk)
6.  {
       这里的 hashinfo 就是 tcp_hashinfo。
7.      struct inet_hashinfo *hinfo = death_row->hashinfo;
8.      const unsigned short snum = inet_sk(sk)->num;
9.      struct inet_bind_hashbucket *head;
10.     struct inet_bind_bucket *tb;
11.     int ret;
12.
13.     if (!snum) {
14.         int low = sysctl_local_port_range[0];
15.         int high = sysctl_local_port_range[1];
16.         int range = high - low;
17.         int i;
18.         int port;
19.         static u32 hint;
20.         u32 offset = hint + inet_sk_port_offset(sk);
21.         struct hlist_node *node;
22.         struct inet_timewait_sock *tw = NULL;
23.
24.         for (i = 1; i <= range; i++) {
25.             port = low + (i + offset) % range;
26.             head = &hinfo->bhash[inet_bhashfn(port, hinfo->bhash_size)];
27.
28.             inet_bind_bucket_for_each(tb, node, &head->chain) {
29.                 if (tb->port == port) {
30.                     BUG_TRAP(!hlist_empty(&tb->owners));
31.                     if (tb->fastreuse >= 0)
32.                         goto next_port;
33.                     if (!__inet_check_established(death_row,
```

```
34.                             sk, port,
35.                             &tw))
36.                         goto ok;
37.                     goto next_port;
38.                 }
39.             }
40.
41.             tb = inet_bind_bucket_create(hinfo->bind_bucket_cachep, head, port);
42.             if (!tb) {
43.                 break;
44.             }
45.             tb->fastreuse = -1;
46.             goto ok;
47.
48.     next_port:
49.             ......
50.         }
51.
52.         return -EADDRNOTAVAIL;
53.
54. ok:
55.         hint += i;
        把自己加入到 hash 表中
56.         inet_bind_hash(sk, tb, port);
57.        if (sk_unhashed(sk)) {
58.             inet_sk(sk)->sport = htons(port);
59.             __inet_hash(hinfo, sk, 0);
60.         }
61.
62.         if (tw) {
63.             inet_twsk_deschedule(tw, death_row);
64.             inet_twsk_put(tw);
65.         }
66.
67.        ret = 0;
68.        goto out;到此处，为此 socket 分配 port 的任务基本完成，可以退出了
69.     }
        下面是当 snum 不为 0 的情况，这必须要检查 snum 的合法性
70.     head = &hinfo->bhash[inet_bhashfn(snum, hinfo->bhash_size)];
71.     tb  = inet_csk(sk)->icsk_bind_hash;
72.     if (sk_head(&tb->owners) == sk && !sk->sk_bind_node.next) {
           注意这里的第三个参数是 0，这表明我们将会把这个 sock 放入 established 表中，而不是
           listening_hash 表中。我们曾经在 tcp_v4_hash 函数中调用过这个函数，当时传入的是 1。
73.        __inet_hash(hinfo, sk, 0);
74.        return 0;
75.    } else {
76.         /* 没有明确的回答，要遍历整个 established hash 表 */
77.        ret = __inet_check_established(death_row, sk, snum, NULL);
78. out:
79.        return ret;
80.    }
81. }
```

代码段 5-17 inet_hash_connect 函数

下面的 ip_route_newports 用处是根据我们本地端口和端口是否与路由查询表中的相同来决定新建一个路由键值，并且调整路由表。它也是 ip_route_output_flow 的封装函数。

```
1.  static inline int ip_route_newports(struct rtable **rp, u16 sport, u16 dport,
2.                  struct sock *sk)
3.  {
4.      if (sport != (*rp)->fl.fl_ip_sport || dport != (*rp)->fl.fl_ip_dport) {
5.          struct flowi fl;
6.          memcpy(&fl, &(*rp)->fl, sizeof(fl));
7.          fl.fl_ip_sport = sport;
8.          fl.fl_ip_dport = dport;
9.
10.          *rp = NULL;
```

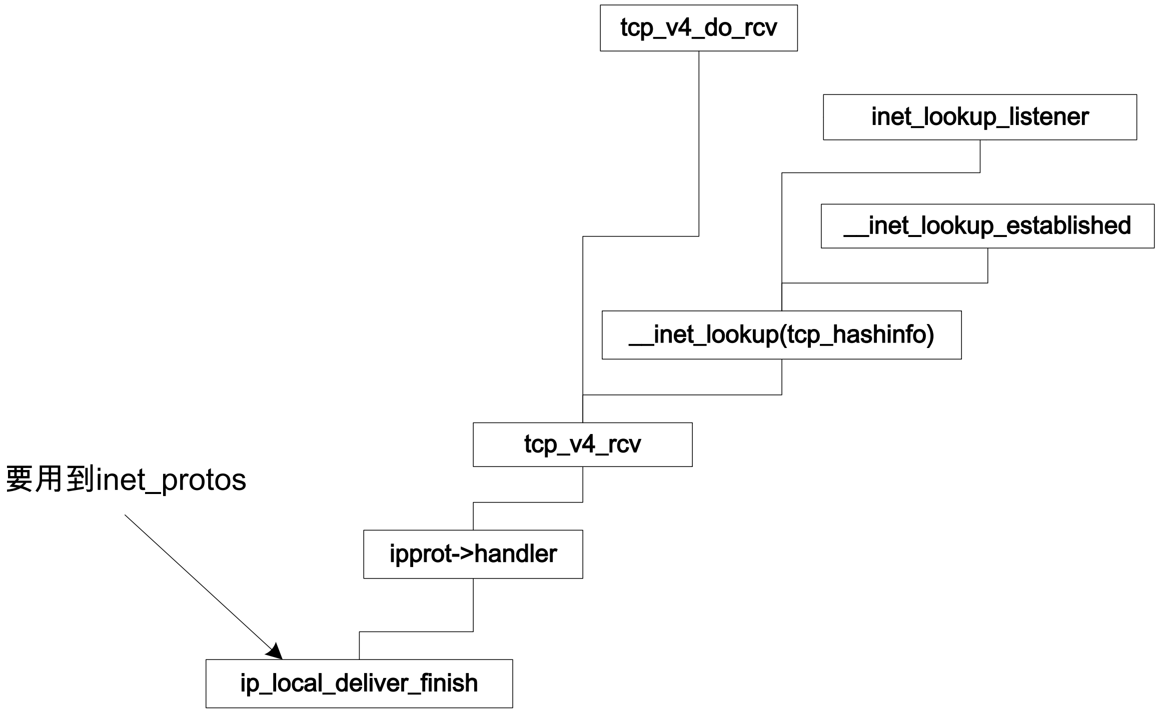
```
11.         return ip_route_output_flow(rp, &fl, sk, 0);
12.     }
13.     return 0;
14. }
```

代码段 5-18 ip_route_newports 函数

5.2.7 TCP 报文的接收

在真正进入三次握手之前，我们必须先说 TCP 报文的接收。

还记得有一个 inet_protocol 结构的变量 tcp_protocol 吗？它的 handler 就是 TCP 层的 tcp_v4_rcv。



图表 5-13ip_local_deliver_finish 函数调用 tcp 的树

如同上一节介绍的 UDP 报文接受方法，TCP 通过搜索 tcp_hashinfo 查找已经建立的 sock{}，不过出于性能的考虑，凡是出于 TCP_ESTABLISHED 状态的 sock{}放入了 tcp_hashinfo→ehash 表中，而处于 TCP_LISTEN 状态的 sock{}放入了 tcp_hashinfo→lhash 表中。

tcp_v4_do_rcv 的代码分析：

```
1.  /* The socket must have it's spinlock held when we get
2.   * here.
3.   *
4.   * We have a potential double-lock case here, so even when
5.   * doing backlog processing we use the BH locking scheme.
6.   * This is because we cannot sleep with the original spinlock
7.   * held.
8.   */
9.  int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
10. {
11.     if (sk->sk_state == TCP_ESTABLISHED)
12.     { /* 快速路径 */
```

```
13.         if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
14.             goto reset;
15.
16.         return 0;
17.     }
18.
19.     if (skb->len < (skb->h.th->doff << 2) || tcp_checksum_complete(skb))
20.         goto csum_err;
21.
22.     if (sk->sk_state == TCP_LISTEN) {
23.         struct sock *nsk = tcp_v4_hnd_req(sk, skb);
24.
25.         if (nsk != sk) {
26.             if (tcp_child_process(sk, nsk, skb))
27.                 goto reset;
28.             return 0;
29.         }
30.     }
31.
32.
33.     if (tcp_rcv_state_process(sk, skb, skb->h.th, skb->len))
34.         goto reset;
35.
36.     return 0;
37.
38. reset:
39.     tcp_v4_send_reset(skb);
40. discard:
41.     kfree_skb(skb);
42.
43.     return 0;
44. csum_err:
45.     goto discard;
46. }
```

代码段 5-19tcp_v4_do_rcv 函数

tcp_rcv_established 的分析：

```
1.  /*
2.   *  TCP 处理 ESTABLISHED 状态报文的接收函数
3.   *
4.   *  此函数被分为“快速路径”和“慢速路径”两部分。快速路径在如下情况被 disabled:
5.   *  - A zero window was announced from us - zero window probing
6.   *       is only handled properly in the slow path.
7.   *  - Out of order segments arrived.
8.   *  - Urgent data is expected.
9.   *  - There is no buffer space left
10.  *  - Unexpected TCP flags/window values/header lengths are received
11.  *    (detected by checking the TCP header against pred_flags)
12.  *  - Data is sent in both directions. Fast path only supports pure senders
13.  *    or pure receivers (this means either the sequence number or the ack
14.  *    value must stay constant)
15.  *  - Unexpected TCP option.
16.  *
17.  *  When these conditions are not satisfied it drops into a standard
18.  *  receive procedure patterned after RFC793 to handle all cases.
19.  *  The first three cases are guaranteed by proper pred_flags setting,
20.  *  the rest is checked inline. Fast processing is turned on in
21.  *  tcp_data_queue when everything is OK.
22.  */
23. int tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
24.         struct tcphdr *th, unsigned len)
25. {
26.         struct tcp_opt *tp = tcp_sk(sk);
27.
28.     /*
29.      *  Header prediction.
30.      *  The code loosely follows the one in the famous
```

```
31.      *  "30 instruction TCP receive" Van Jacobson mail.
32.      *
33.      *  Van's trick is to deposit buffers into socket queue
34.      *  on a device interrupt, to call tcp_recv function
35.      *  on the receive process context and checksum and copy
36.      *  the buffer to user space. smart...
37.      *
38.      *  Our current scheme is not silly either but we take the
39.      *  extra cost of the net_bh soft interrupt processing...
40.      *  We do checksum and copy also but from device to kernel.
41.      */
42.
43.     tp->saw_tstamp = 0;
44.
45.     /*  pred_flags is 0xS?10 << 16 + snd_wnd
46.      *  if header_predition is to be made
47.      *  'S' will always be tp->tcp_header_len >> 2
48.      *  '?' will be 0 for the fast path, otherwise pred_flags is 0 to
49.      *  turn it off (when there are holes in the receive
50.      *   space for instance)
51.      *  PSH flag is ignored.
52.      */
53.
54.     if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
55.         TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
56.         int tcp_header_len = tp->tcp_header_len;
57.
58.         /* Timestamp header prediction: tcp_header_len
59.          * is automatically equal to th->doff*4 due to pred_flags
60.          * match.
61.          */
62.
63.         /* Check timestamp */
64.         if (tcp_header_len ==sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) {
65.             __u32 *ptr = (__u32 *)(th + 1);
66.
67.             /* No? Slow path! */
68.             if (*ptr != ntohl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16)
69.                       | (TCPOPT_TIMESTAMP << 8) | TCPOLEN_TIMESTAMP))
70.                 goto slow_path;
71.
72.             tp->saw_tstamp = 1;
73.             ++ptr;
74.             tp->rcv_tsval = ntohl(*ptr);
75.             ++ptr;
76.             tp->rcv_tsecr = ntohl(*ptr);
77.
78.             /* If PAWS failed, check it more carefully in slow path */
79.             if ((s32)(tp->rcv_tsval - tp->ts_recent) < 0)
80.                 goto slow_path;
81.
82.             /* DO NOT update ts_recent here, if checksum fails
83.              * and timestamp was corrupted part, it will result
84.              * in a hung connection since we will drop all
85.              * future packets due to the PAWS test.
86.              */
87.         }
88.         if (len <= tcp_header_len) {
89.            /* Bulk data transfer: sender */
90.            if (len == tcp_header_len) {
91.             /* Predicted packet is in window by definition.
92.              * seq == rcv_nxt and rcv_wup <= rcv_nxt.
93.              * Hence, check seq<=rcv_wup reduces to:
94.              */
95.                if (tcp_header_len ==
96.                    (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
97.                    tp->rcv_nxt == tp->rcv_wup)
98.                     tcp_store_ts_recent(tp);
99.                /* We know that such packets are checksummed on entry. */
```

```
100.                    tcp_ack(sk, skb, 0);
101.                    __kfree_skb(skb);
102.                    tcp_data_snd_check(sk);
103.                    return 0;
104.                } else { /* Header too small */
105.                    TCP_INC_STATS_BH(TcpInErrs);
106.                    goto discard;
107.                }
108.            } else {
109.                int eaten = 0;
110.
111.                if (tp->ucopy.task == current &&
112.                    tp->copied_seq == tp->rcv_nxt &&
113.                    len - tcp_header_len <= tp->ucopy.len &&
114.                    sock_owned_by_user(sk)) {
115.                    __set_current_state(TASK_RUNNING);
116.
117.                    if (!tcp_copy_to_iovec(sk, skb, tcp_header_len)) {
118.                        /* Predicted packet is in window by definition.
119.                         * seq == rcv_nxt and rcv_wup <= rcv_nxt.
120.                         * Hence, check seq<=rcv_wup reduces to:
121.                         */
122.                        if (tcp_header_len ==
123.                            (sizeof(struct tcphdr) +
124.                             TCPOLEN_TSTAMP_ALIGNED) &&
125.                            tp->rcv_nxt == tp->rcv_wup)
126.                             tcp_store_ts_recent(tp);
127.
128.                        __skb_pull(skb, tcp_header_len);
129.                        tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
130.                        eaten = 1;
131.                    }
132.                }
133.                if (!eaten) {
134.                    if (tcp_checksum_complete_user(sk, skb))
135.                        goto csum_error;
136.
137.                    /* Predicted packet is in window by definition.
138.                     * seq == rcv_nxt and rcv_wup <= rcv_nxt.
139.                     * Hence, check seq<=rcv_wup reduces to:
140.                     */
141.                    if (tcp_header_len ==
142.                        (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
143.                        tp->rcv_nxt == tp->rcv_wup)
144.                        tcp_store_ts_recent(tp);
145.
146.                    if ((int)skb->truesize > sk->sk_forward_alloc)
147.                        goto step5;
148.
149.                    NET_INC_STATS_BH(TCPHPHits);
150.
151.                    /* Bulk data transfer: receiver */
152.                    __skb_pull(skb,tcp_header_len);
153.                    __skb_queue_tail(&sk->sk_receive_queue, skb);
154.                    tcp_set_owner_r(skb, sk);
155.                    tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
156.                }
157.
158.                tcp_event_data_recv(sk, tp, skb);
159.
160.                if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
161.                    /* Well, only one small jumplet in fast path... */
162.                    tcp_ack(sk, skb, FLAG_DATA);
163.                    tcp_data_snd_check(sk);
164.                    if (!tcp_ack_scheduled(tp))
165.                        goto no_ack;
166.                }
167.
168.                if (eaten) {
```

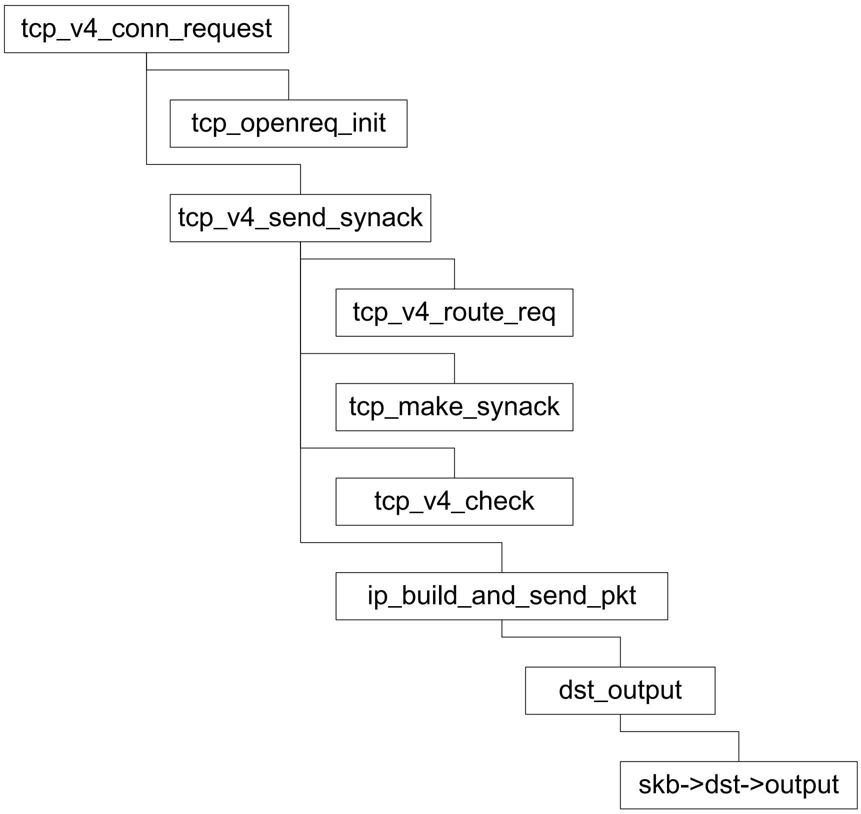
```
169.                    if (tcp_in_quickack_mode(tp)) {
170.                        tcp_send_ack(sk);
171.                    } else {
172.                        tcp_send_delayed_ack(sk);
173.                    }
174.                } else {
175.                    __tcp_ack_snd_check(sk, 0);
176.                }
177.
178.    no_ack:
179.                if (eaten)
180.                    __kfree_skb(skb);
181.                else
182.                    sk->sk_data_ready(sk, 0);
183.                return 0;
184.            }
185.        }
186.
187.    slow_path:
188.        if (len < (th->doff<<2) || tcp_checksum_complete_user(sk, skb))
189.            goto csum_error;
190.
191.        /*
192.         * RFC1323: H1. Apply PAWS check first.
193.         */
194.        if (tcp_fast_parse_options(skb, th, tp) && tp->saw_tstamp &&
195.           tcp_paws_discard(tp, skb)) {
196.            if (!th->rst) {
197.                NET_INC_STATS_BH(PAWSEstabRejected);
198.                tcp_send_dupack(sk, skb);
199.                goto discard;
200.            }
201.            /* Resets are accepted even if PAWS failed.
202.
203.               ts_recent update must be made after we are sure
204.               that the packet is in window.
205.             */
206.        }
207.
208.        /*
209.         *  Standard slow path.
210.         */
211.
212.      if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)){
213.            ......
214.            if (!th->rst)
215.                tcp_send_dupack(sk, skb);
216.            goto discard;
217.        }
218.
219.        if(th->rst) {
220.            tcp_reset(sk);
221.            goto discard;
222.        }
223.
224.        tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
225.
226.        if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
227.            TCP_INC_STATS_BH(TcpInErrs);
228.            NET_INC_STATS_BH(TCPAbortOnSyn);
229.            tcp_reset(sk);
230.            return 1;
231.        }
232.
233.    step5:
234.        if(th->ack)
235.            tcp_ack(sk, skb, FLAG_SLOWPATH);
236.
237.        /* Process urgent data. */
```

```
238.        tcp_urg(sk, skb, th);
239.
240.        /* step 7: process the segment text */
241.        tcp_data_queue(sk, skb);
242.
243.        tcp_data_snd_check(sk);
244.        tcp_ack_snd_check(sk);
245.        return 0;
246.
247.    csum_error:
248.    ......
249.    discard:
250.        __kfree_skb(skb);
251.        return 0;
252.    }
```

代码段 5-20tcp_rcv_established 函数

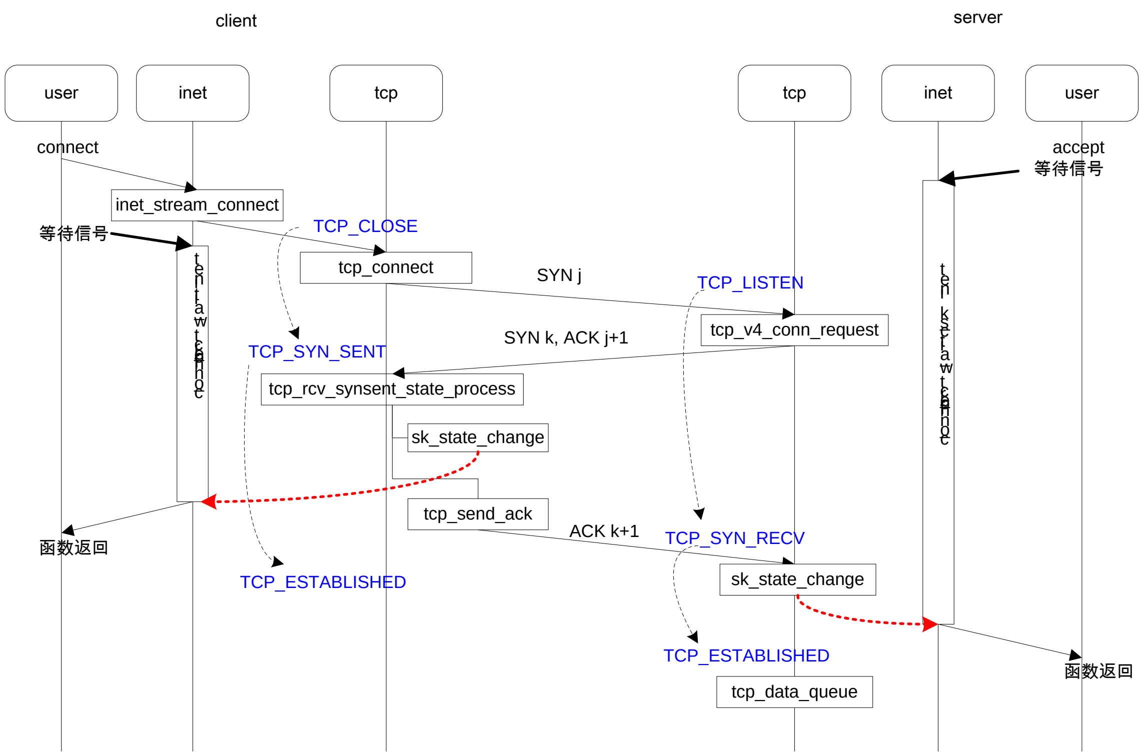
5.2.8　3 次握手的实现

不管是客户端还是服务器端，当连接不是 TCP_ESTABLISHED 状态的时候，就会调用 tcp_rcv_state_process 函数，这就是专门处理 3 次握手协议的状态机处理函数。对于服务器端 TCP 协议栈，当收到一个 connect 请求时，会去调用此函数中出现的 tp->af_specific->**conn_request**(sk, skb)，也就是 tcp_v4_conn_request，如果不记得如何指向此函数，请参考 TCP 的 socket 初始化：



图表 5-14tcp_v4_conn_request 函数调用树

在此，skb->dst->output 要根据具体的情况分析，不过，一般都是 ip_output。我们在上面的发送流程分析中已经介绍



图表 5-15 3 次握手在内核中的实现序列图

当一台主机向另一台主机发起 tcp 连接请求时，它所发出的第一个 tcp 数据报是一个 SYN，并带上自己的 ISN(初始序号)。

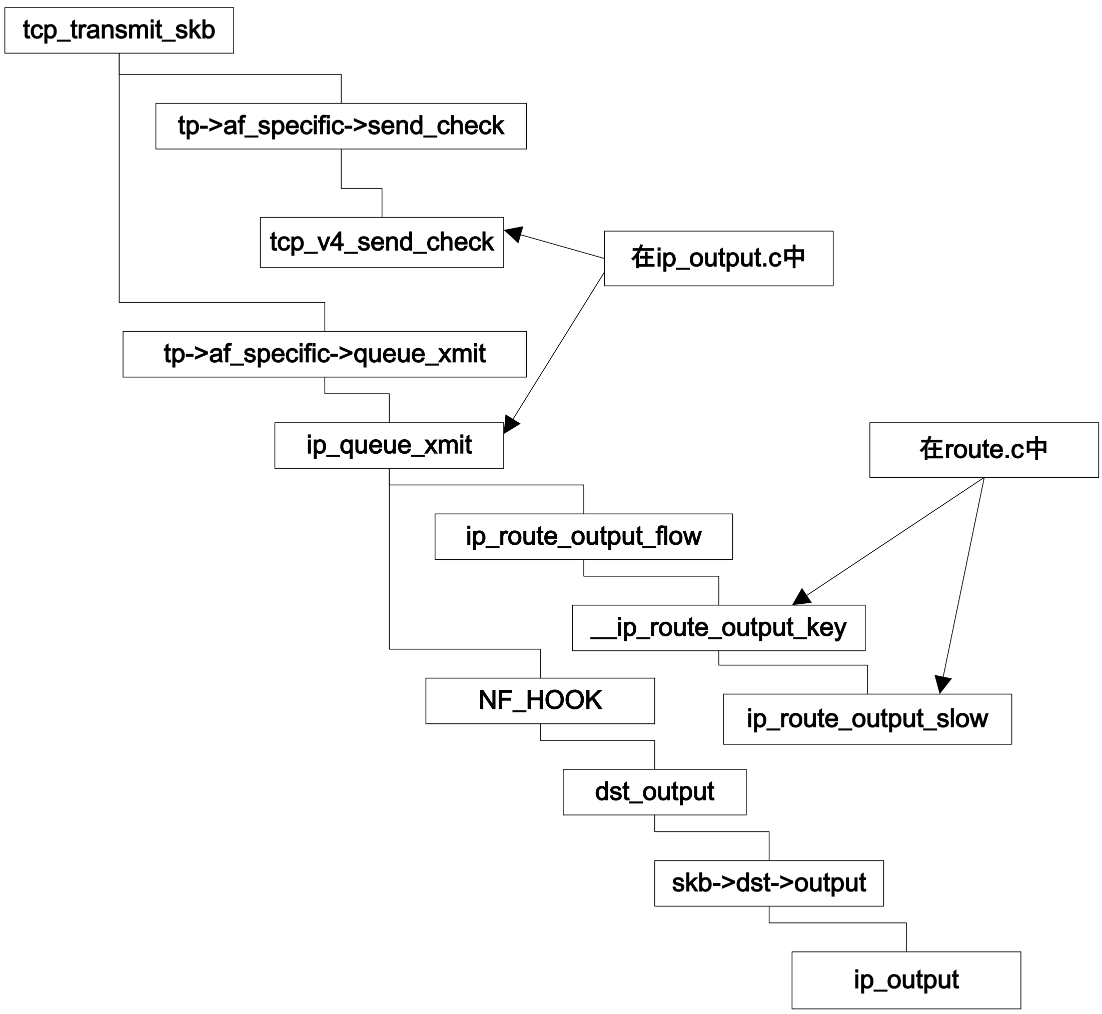
当一个 TCP 报文送到协议栈的时候，首先在 ehash 表的前半部分中查寻，即在处于 TCP_ESTABLISH 状态的 socket 中查寻匹配。匹配逻辑是：socket 结构体成员 sk_hash 要等于根据请求数据报的目的地址和源地址等信息计算出来的哈希值，数据报的目的地址等于本地 socket 的接收源地址，数据报的源地址等于本地 socket 的目的地址，数据的源端口和目的端口与本地 socket 的目的端口与源端口匹配，并且，如果本地 socket 绑定了输入网络设备接口，那数据报的输入接口要跟本地 socket 的输入接口相同。

因为这是一个连接请求，所以在 ehash 表中是不可能找到匹配的 socket 的。接下来到 listening_hash 中寻找侦听 socket。只要目的地址和端口匹配，如果侦听 socket 绑定了输入接口，输入接口也匹配，则寻找成功。找到一个侦听 socket 后，首先在这个 socket 的 accept 队列(struct inet_connection_sock->icsk_accept_queue)中寻找匹配的 request_sock，因为有可能该请求已经被接受，并保存在接受队列中了（但还没有被 accept 处理，因为如果 accept 处理过了，会出现在 ehash 表中)。寻找 request_sock 的逻辑是请求数据的源地址和端口跟 rmt_addr 和 rmt_port 匹配，目的地址跟 loc_addr 匹配。显然，对于刚刚发起第一个 SYN 的 socket 请求来讲，这种寻找总是失败的，它不可能已经在接受队列中存在。

没有找到任何 socket，我们继续处理这个侦听 socket 本身，它当前处于 TCP_LISTEN 状态，并且收到了来自网络对端的第一个 SYN 数据报，于是调用 tcp_v4_conn_request 接受这个连接请求，tcp_v4_conn_request 函数对于接受 tcp 连接请求的处理，它在创建了一个 struct request_sock 之后，需要生成一个 ISN(初始序号)。有了 ISN 后，对请求 socket 发出的 SYN 发回一个 ACK 跟本地的 SYN。然后把创建的 struct request_sock 加入到侦听 socket 的请求 socket 哈希表中，(struct

inet_connection_sock→isck_accept_queue→listen_opt→syn_table)>，但是需要注意的是，此时该 request_sock 还没有进入接受队列(即通过 struct inet_connection_sock → isck_accept_queue →rskq_accept_head 还不能找到它)。

接下来，当客户端的 socket 收到来自本地 socket 的 ACK+SYN 后，会发回来一个 ACK。继续走 tcp 数据报的接收流程。此时我们还是只能在 tcp_hashinfo 哈希表集中的 listening_hash 表中找到侦听 socket 对它进行处理。但此时，我们可以在侦听 socket 的 syn_table 表中找到这个 request_sock。找出这个 request_sock 后，要对它进行检查，检查通过，从侦听 socket 上新克隆一个 strcut tcp_sock，置其状态为 TCP_SYN_RECV，放入 mytcp_hashinfok 中的 ehash 哈希表，然后把该新创建的 tcp_sock 也绑定在跟侦听端口相同的本地端口上。最后，把 socket 的状态改为 TCP_ESTABLISH，接受一个 tcp 连接请求的过程结束。accept 系统调用把 socket 从接受队列中取走，取回 request_sock->sk，并释放掉 request_sock，然后就可以进行 tcp 通讯了。



图表 5-16tcp_transmit_skb 调用树

5.2.9 内核收到报文转到用户态

先说说 TCP 收包的 context（不定长包）。一般情况，发送方发送一个包，然后接收方收到一个包，这是最好处理的。第二种情况，当每次发生的包比较小时，发送数据时，TCP 会启用优化

算法，将多个小包集中起来发送，以提高传输效率。此时接收方的 recv buffer 中，可能出现不止一个包。第三种情况，recv buffer 中每次只一个包，但接收方没及时取包，这时 recv buffer 中会积累多个包。

理所当然，TCP 收包要考虑所有这些情况。一般来说有三种方法。第一种，定义好通讯协议，先收包头，然后根据包头中的消息真实大小，接收消息剩余部分。第二种方法，通讯协议规定好每个消息的开始和结束标识符。然后每次 recv 得到的数据先放到一个大（比如你的最大 packet 的 2 倍）buffer 中，最后再来分析这个 buffer 分包。第三种方法，先用 recv+MSG_PEEK 接收某个固定长度，然后对接收到的"包"进行分析，然后做真正的 recv 操作。MSG_PEEK 标志位的作用在于，用户进程收到数据以后，并不从缓冲区中删除，再次执行 recv 的时候，依然能够收到数据，这种情况下，把当前的 seq 传给 peek_seq,再把 seq 指向 peek_seq

```
1.  /*
2.   *  这个函数把 sock 中的数据拷贝到用户缓冲区*
3.   *  Technical note: in 2.3 we work on _locked_ socket, so that
4.   *  tricks with *seq access order and skb->users are not required.
5.   *  Probably, code can be easily improved even more.
6.   */
7.
8.  int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
9.      size_t len, int nonblock, int flags, int *addr_len)
10. {
11.         struct tcp_opt *tp = tcp_sk(sk);
12.     int copied = 0;
13.     u32 peek_seq;
14.     u32 *seq;
15.     unsigned long used;
16.     int err;
17.     int target;     /* Read at least this many bytes */
18.     long timeo;
19.     struct task_struct *user_recv = NULL;
20.
21.     err = -ENOTCONN;
22.     if (sk->sk_state == TCP_LISTEN)
23.         goto out;
24.
25.     timeo = sock_rcvtimeo(sk, nonblock);
26.
27.     /* 带外数据需要特别处理*/
28.     if (flags & MSG_OOB)
29.         goto recv_urg;
30.
31.     seq = &tp->copied_seq;
32.     if (flags & MSG_PEEK) {
33.         peek_seq = tp->copied_seq;
34.         seq = &peek_seq;
35.     }
36.
37.     target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
38.
39.     do {
40.         struct sk_buff *skb;
41.         u32 offset;
42.
            判断是否是紧急数据?如果我们已经读取了一些数据或者还有 SIGURG 信号没有处理的话则必
            须停下来
43.         if (tp->urg_data && tp->urg_seq == *seq) {
44.             if (copied)
45.                 break;
46.             if (signal_pending(current)) {
47.                 copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
48.                 break;
49.             }
```

```
50.         }
        从接收队列中获取一个 skb
51.     skb = skb_peek(&sk->sk_receive_queue);
52.     do {
53.         if (!skb)
54.             break;
55.
56.         /* Now that we have two receive queues this
57.          * shouldn't happen.
58.          */
59.         if (before(*seq, TCP_SKB_CB(skb)->seq)) {
60.
61.             break;
62.         }
63.         offset = *seq - TCP_SKB_CB(skb)->seq;
64.         if (skb->h.th->syn)
65.             offset--;
66.         if (offset < skb->len)
67.             goto found_ok_skb;
68.         if (skb->h.th->fin)
69.             goto found_fin_ok;
70.         BUG_TRAP(flags & MSG_PEEK);
71.         skb = skb->next;
72.     } while (skb != (struct sk_buff *)&sk->sk_receive_queue);
73.
74.     /* 如果我们有 backlog 设备，就尝试用它来处理
75.
76.     if (copied >= target && !sk->sk_backlog.tail)
77.         break;
78.
79.     if (copied) {
80.         if (sk->sk_err ||
81.             sk->sk_state == TCP_CLOSE ||
82.             (sk->sk_shutdown & RCV_SHUTDOWN) ||
83.             !timeo ||
84.             signal_pending(current) ||
85.             (flags & MSG_PEEK))
86.             break;
87.     } else {
88.         if (sock_flag(sk, SOCK_DONE))
89.             break;
90.         ......
91.
92.         if (sk->sk_shutdown & RCV_SHUTDOWN)
93.             break;
94.
95. if (sk->sk_state == TCP_CLOSE) {
96.     if (!sock_flag(sk, SOCK_DONE)) {
97.         /* This occurs when user tries to read
98.          * from never connected socket.
99.          */
100.                copied = -ENOTCONN;
101.                break;
102.            }
103.            break;
104.        }
105.
106.        if (!timeo) {
107.            copied = -EAGAIN;
108.            break;
109.        }
110.
111.        if (signal_pending(current)) {
112.            copied = sock_intr_errno(timeo);
113.            break;
114.        }
115.    }
116.
117.    cleanup_rbuf(sk, copied);
```

```
118.
119.    if (tp->ucopy.task == user_recv) {
120.        /* Install new reader */
121.        if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
122.            user_recv = current;
123.            tp->ucopy.task = user_recv;
124.            tp->ucopy.iov = msg->msg_iov;
125.        }
126.        tp->ucopy.len = len;
127.
128.        /* Ugly... If prequeue is not empty, we have to
129.         * process it before releasing socket, otherwise
130.         * order will be broken at second iteration.
131.         * More elegant solution is required!!!
132.         *
133.         * Look: we have the following (pseudo)queues:
134.         *
135.         * 1. packets in flight
136.         * 2. backlog
137.         * 3. prequeue
138.         * 4. receive_queue
139.         *
140.         * Each queue can be processed only if the next ones
141.         * are empty. At this point we have empty receive_queue.
142.         * But prequeue _can_ be not empty after 2nd iteration,
143.         * when we jumped to start of loop because backlog
144.         * processing added something to receive_queue.
145.         * We cannot release_sock(), because backlog contains
146.         * packets arrived _after_ prequeued ones.
147.         *
148.         * Shortly, algorithm is clear --- to process all
149.         * the queues in order. We could make it more directly,
150.         * requeueing packets from backlog to prequeue, if
151.         * is not empty. It is more elegant, but eats cycles,
152.         * unfortunately.
153.         */
154.        if (skb_queue_len(&tp->ucopy.prequeue))
155.            goto do_prequeue;
156.        /* __ Set realtime policy in scheduler __ */
157.    }
158.    if (copied >= target) {
159.        /* Do not sleep, just process backlog. */
160.        release_sock(sk);
161.        } else {
162.        timeo = tcp_data_wait(sk, timeo);
163.    }
164.    if (user_recv) {
165.        int chunk;
166.        /* __ Restore normal policy in scheduler __ */
167.
168.        if ((chunk = len - tp->ucopy.len) != 0) {
169.            len -= chunk;
170.            copied += chunk;
171.        }
172.
173.    if (tp->rcv_nxt == tp->copied_seq &&
174.            skb_queue_len(&tp->ucopy.prequeue)) {
175.    do_prequeue:
176.            tcp_prequeue_process(sk);
177.            if ((chunk = len - tp->ucopy.len) != 0) {
178.                len -= chunk;
179.                copied += chunk;
180.            }
181.        }
182.    }
183.    if ((flags & MSG_PEEK) && peek_seq != tp->copied_seq) {
184.        current->comm, current->pid);
185.        peek_seq = tp->copied_seq;
186.    }
```

```
187.    continue;
188.
189.    found_ok_skb:
190.            /* Ok so how much can we use? */
191.            used = skb->len - offset;
192.            if (len < used)
193.                used = len;
194.
195.            /* Do we have urgent data here? */
196.            if (tp->urg_data) {
197.                u32 urg_offset = tp->urg_seq - *seq;
198.                if (urg_offset < used) {
199.                    if (!urg_offset) {
200.                        if (!sock_flag(sk, SOCK_URGINLINE)) {
201.                            ++*seq;
202.                            offset++;
203.                            used--;
204.                            if (!used)
205.                                goto skip_copy;
206.                        }
207.                    } else
208.                        used = urg_offset;
209.                }
210.            }
211.
212.            if (!(flags & MSG_TRUNC)) {
213.                err = skb_copy_datagram_iovec(skb, offset,
214.                                    msg->msg_iov, used);
215.                if (err) {
216.                    ......
217.                    break;
218.                }
219.            }
220.
221.            *seq += used;
222.            copied += used;
223.            len -= used;
224.
225.    skip_copy:
226.            if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
227.                tp->urg_data = 0;
228.                tcp_fast_path_check(sk, tp);
229.            }
230.            if (used + offset < skb->len)
231.                continue;
232.
233.            if (skb->h.th->fin)
234.                goto found_fin_ok;
235.            if (!(flags & MSG_PEEK))
236.                tcp_eat_skb(sk, skb);
237.            continue;
238.
239.        found_fin_ok:
240.            /* Process the FIN. */
241.            ++*seq;
242.            if (!(flags & MSG_PEEK))
243.                tcp_eat_skb(sk, skb);
244.            break;
245.        } while (len > 0);
246.
247.        if (user_recv) {
248.            if (skb_queue_len(&tp->ucopy.prequeue)) {
249.                int chunk;
250.
251.                tp->ucopy.len = copied > 0 ? len : 0;
252.
253.                tcp_prequeue_process(sk);
254.
255.                if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
```

```
256.                    len -= chunk;
257.                    copied += chunk;
258.                }
259.            }
260. 
261.            tp->ucopy.task = NULL;
262.            tp->ucopy.len = 0;
263.        }
264. 
265.        /* Clean up data we have read: This will do ACK frames. */
266.        cleanup_rbuf(sk, copied);
267. 
268.        release_sock(sk);
269.        return copied;
270. 
271.    out:
272.        release_sock(sk);
273.        return err;
274. 
275.    recv_urg:
276.        err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
277.        goto out;
278.    }
```

代码段 5-21 tcp_recvmsg

5.2.10 释放 TCP 的 socket

当要关闭 TCP 的 socket 时要进行很多判断，主要是连接状态检查，这与 UDP、RAW IP 有很大不同。

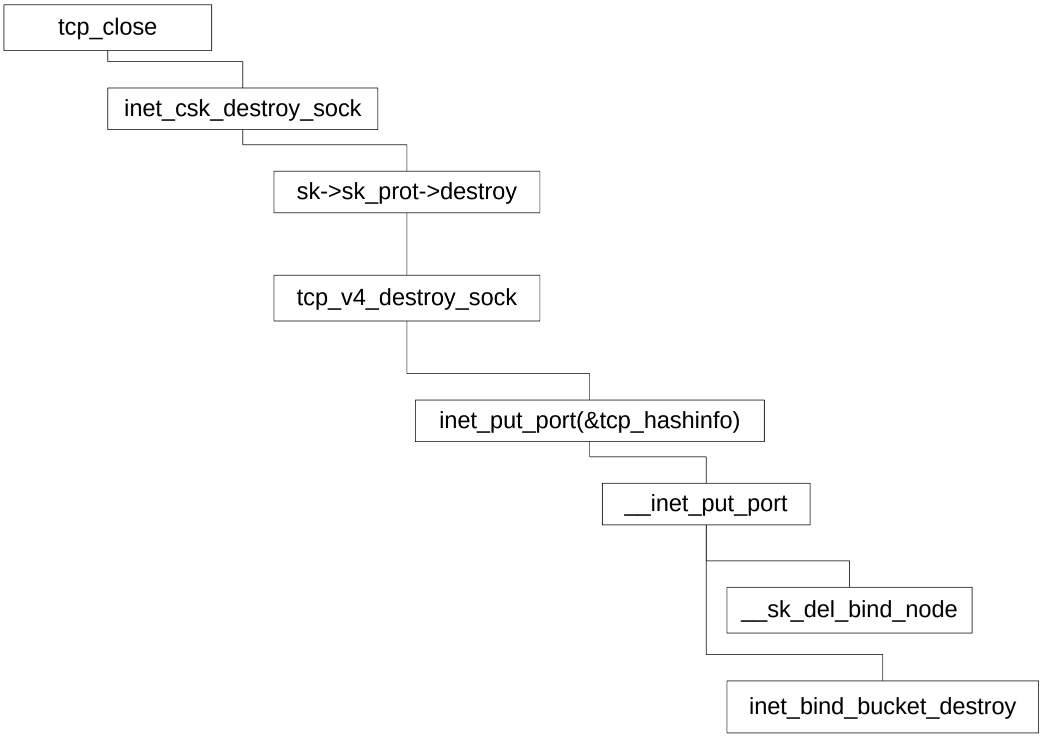
```
1.  void tcp_close(struct sock *sk, long timeout)
2.  {
3.      struct sk_buff *skb;
4.      int data_was_unread = 0;
5.      int state;
6. 
7.      lock_sock(sk);
8.      sk->sk_shutdown = SHUTDOWN_MASK;
9. 
10.     if (sk->sk_state == TCP_LISTEN) {
11.         tcp_set_state(sk, TCP_CLOSE);
12. 
13.         /* Special case. */
14.         inet_csk_listen_stop(sk);
15. 
16.         goto adjudge_to_death;
17.     }
18. 
19.     /*  We need to flush the recv. buffs.  We do this only on the
20.      *  descriptor close, not protocol-sourced closes, because the
21.      *  reader process may not have drained the data yet!
22.      */
23.     while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {
24.         u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq -
25.               skb->h.th->fin;
26.         data_was_unread += len;
27.         __kfree_skb(skb);
28.     }
29. 
30.     sk_stream_mem_reclaim(sk);
31. 
32.     /* As outlined in draft-ietf-tcpimpl-prob-03.txt, section
33.      * 3.10, we send a RST here because data was lost.  To
34.      * witness the awful effects of the old behavior of always
35.      * doing a FIN, run an older 2.1.x kernel or 2.0.x, start
```

```
36.       * a bulk GET in an FTP client, suspend the process, wait
37.       * for the client to advertise a zero window, then kill -9
38.       * the FTP client, wheee...  Note: timeout is always zero
39.       * in such a case.
40.       */
41.      if (data_was_unread) {
42.          /* Unread data was tossed, zap the connection. */
43.          NET_INC_STATS_USER(LINUX_MIB_TCPABORTONCLOSE);
44.          tcp_set_state(sk, TCP_CLOSE);
45.          tcp_send_active_reset(sk, GFP_KERNEL);
46.      } else if (sock_flag(sk, SOCK_LINGER) && !sk->sk_lingertime) {
47.          /* Check zero linger _after_ checking for unread data. */
48.          sk->sk_prot->disconnect(sk, 0);
49.          NET_INC_STATS_USER(LINUX_MIB_TCPABORTONDATA);
50.      } else if (tcp_close_state(sk)) {
51.          /* We FIN if the application ate all the data before
52.           * zapping the connection.
53.           */
54.
55.          /* RED-PEN. Formally speaking, we have broken TCP state
56.           * machine. State transitions:
57.           *
58.           * TCP_ESTABLISHED -> TCP_FIN_WAIT1
59.           * TCP_SYN_RECV -> TCP_FIN_WAIT1 (forget it, it's impossible)
60.           * TCP_CLOSE_WAIT -> TCP_LAST_ACK
61.           *
62.           * are legal only when FIN has been sent (i.e. in window),
63.           * rather than queued out of window. Purists blame.
64.           *
65.           * F.e. "RFC state" is ESTABLISHED,
66.           * if Linux state is FIN-WAIT-1, but FIN is still not sent.
67.           *
68.           * The visible declinations are that sometimes
69.           * we enter time-wait state, when it is not required really
70.           * (harmless), do not send active resets, when they are
71.           * required by specs (TCP_ESTABLISHED, TCP_CLOSE_WAIT, when
72.           * they look as CLOSING or LAST_ACK for Linux)
73.           * Probably, I missed some more holelets.
74.           *                      --ANK
75.           */
76.          tcp_send_fin(sk);
77.      }
78.
79.      sk_stream_wait_close(sk, timeout);
80.
81. adjudge_to_death:
82.      state = sk->sk_state;
83.      sock_hold(sk);
84.      sock_orphan(sk);
85.      atomic_inc(sk->sk_prot->orphan_count);
86.
87.      /* It is the last release_sock in its life. It will remove backlog. */
88.      release_sock(sk);
89.
90.
91.      /* Now socket is owned by kernel and we acquire BH lock
92.         to finish close. No need to check for user refs.
93.       */
94.      local_bh_disable();
95.      bh_lock_sock(sk);
96.
97.      /* Have we already been destroyed by a softirq or backlog? */
98.      if (state != TCP_CLOSE && sk->sk_state == TCP_CLOSE)
99.          goto out;
100.
101.             /*  This is a (useful) BSD violating of the RFC. There is a
102.              *  problem with TCP as specified in that the other end could
103.              *  keep a socket open forever with no application left this end.
104.              *  We use a 3 minute timeout (about the same as BSD) then kill
```

```
105.             *  our end. If they send after that then tough - BUT: long enough
106.             *  that we won't make the old 4*rto = almost no time - whoops
107.             *  reset mistake.
108.             *
109.             *  Nope, it was not mistake. It is really desired behaviour
110.             *  f.e. on http servers, when such sockets are useless, but
111.             *  consume significant resources. Let's do it with special
112.             *  linger2 option.                --ANK
113.             */
114.
115.            if (sk->sk_state == TCP_FIN_WAIT2) {
116.                struct tcp_sock *tp = tcp_sk(sk);
117.                if (tp->linger2 < 0) {
118.                    tcp_set_state(sk, TCP_CLOSE);
119.                    tcp_send_active_reset(sk, GFP_ATOMIC);
120.                    NET_INC_STATS_BH(LINUX_MIB_TCPABORTONLINGER);
121.                } else {
122.                    const int tmo = tcp_fin_time(sk);
123.
124.                    if (tmo > TCP_TIMEWAIT_LEN) {
125.                        inet_csk_reset_keepalive_timer(sk,
126.                                tmo - TCP_TIMEWAIT_LEN);
127.                    } else {
128.                        tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
129.                        goto out;
130.                    }
131.                }
132.            }
133.            if (sk->sk_state != TCP_CLOSE) {
134.                sk_stream_mem_reclaim(sk);
135.                if (atomic_read(sk->sk_prot->orphan_count) >
    sysctl_tcp_max_orphans ||
136.                    (sk->sk_wmem_queued > SOCK_MIN_SNDBUF &&
137.                     atomic_read(&tcp_memory_allocated) > sysctl_tcp_mem[2])) {
138.                    if (net_ratelimit())
139.                        printk(KERN_INFO "TCP: too many of orphaned "
140.                              "sockets\n");
141.                    tcp_set_state(sk, TCP_CLOSE);
142.                    tcp_send_active_reset(sk, GFP_ATOMIC);
143.                    NET_INC_STATS_BH(LINUX_MIB_TCPABORTONMEMORY);
144.                }
145.            }
146.
147.            if (sk->sk_state == TCP_CLOSE)
148.                inet_csk_destroy_sock(sk);
149.            /* Otherwise, socket is reprieved until protocol close. */
150.
151.        out:
152.            bh_unlock_sock(sk);
153.            local_bh_enable();
154.            sock_put(sk);
155.    }
```

代码段 5-22 tcp_close 函数

把之前对状态的检查去掉，实际到最后忍无可忍要关闭 socket 时，也就是调用了 inet_csk_destroy_sock 函数，通过分析其调用树，最后关键的动作除了清除发送和接收队列，再就是把 tcp_hashinfo 中的 hash 节点删除。



图表 5-17 tcp_close 函数调用树

Socket 系统还提供了一个类似于 close 的函数：int shutdown(int sockfd,int howto)

TCP 连接是双向的(是可读写的),当我们使用 close 时,会把读写通道都关闭,有时侯我们希望只关闭一个方向,这个时候我们可以使用 shutdown.针对不同的 howto,系统回采取不同的关闭方式：

howto=0 这个时候系统会关闭读通道.但是可以继续往接字描述符写.

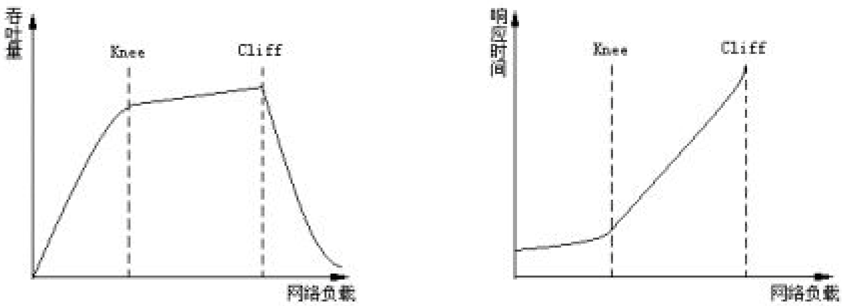
howto=1 关闭写通道,和上面相反,着时候就只可以读了.

howto=2 关闭读写通道,和 close 一样 在多进程程序里面,如果有几个子进程共享一个套接字时,如果我们使用 shutdown, 那么所有的子进程都不能够操作了,这个时候我们只能够使用 close 来关闭子进程的套接字描述符.

至此，关于基本 socket 实现中最关键的部分，已经介绍完毕。有读者说怎么没有具体协议的介绍啊？本书开头不是说了吗，本书不是解说协议本身的，而是介绍协议是如何实现的。这两者有区别的。如果要介绍协议，我完全可以把 RFC 的内容复制过来。但那有什么技术含量呢？

5.3 TCP 拥塞控制

当网络中存在过多的数据包时，网络的性能就会下降，这种现象称为拥塞。在网络发生拥塞时，会导致吞吐量下降，严重时会发生“拥塞崩溃”（congestion collapse）现象。一般来说，拥塞崩溃发生在网络负载的增加导致网络效率的降低的时候。最初观察到这种现象是在 1986 年 10 月，在这个过程中，LBL 与 UC Berkeley 之间的吞吐量从 32kbps 下降到了 40bps。Floyd 总结出拥塞崩溃主要包括以下几种：传统的崩溃、未传送数据包导致的崩溃、由于数据包分段造成的崩溃、日益增长的控制信息流造成的崩溃等。



图表 5-18网络负载与吞吐量及响应时间的关系

对于拥塞现象，我们可以进一步用图 1 来描述。当网络负载较小时，吞吐量基本上随着负载的增长而增长，呈线性关系，响应时间增长缓慢。当负载达到网络容量时，吞吐量呈现出缓慢增长，而响应时间急剧增加，这一点称为 Knee。如果负载继续增加，路由器开始丢包，当负载超过一定量时，吞吐量开始急剧下降，这一点称为 Cliff。拥塞控制机制实际上包含拥塞避免（congestion avoidance）和拥塞控制（congestion control）两种策略。前者的目的是使网络运行在 Knee 附近，避免拥塞的发生；而后者则是使得网络运行在 Cliff 的左侧区域。前者是一种“预防”措施，维持网络的高吞吐量、低延迟状态，避免进入拥塞；后者是一种“恢复”措施，使网络从拥塞中恢复过来，进入正常的运行状态。

拥塞现象的发生和前面提到的互联网的设计机制有着密切关系，我们对这种设计机制作一个简单的归纳：

数据包交换（packet switched）网络：与电路交换（circuit switched）网络相比，由于包交换网络对资源的利用是基于统计复用（statistical multiplexing）的，因此提高了资源的利用效率。但在基于统计复用的情况下，很难保证用户的服务质量（quality of service，QoS），并且很容易出现数据包“乱序”的现象，对乱序数据包的处理会大大增加拥塞控制的复杂性。

无连接（connectionless）网络：互联网的节点之间在发送数据之前不需要建立连接，从而简化了网络的设计，网络的中间节点上无需保留和连接有关的状态信息。但无连接模型很难引入接纳控制（admission control），在用户需求大于网络资源时难以保证服务质量；此外，由于对数据发送源的追踪能力很差，给网络安全带来了隐患；无连接也是网络中出现乱序数据包的主要原因。

“尽力而为”的服务模型：不对网络中传输的数据提供服务质量保证。在这种服务模型下，所有的业务流被“一视同仁”地公平地竞争网络资源，路由器对所有的数据包都采用先来先处理（First Come First Service，FCFS）的工作方式，它尽最大努力将数据包包送达目的地。但对数据包传递

的可靠性、延迟等不能提供任何保证。这很适合 Email、Ftp、WWW 等业务。但随着互联网的飞速发展，IP 业务也得到了快速增长和多样化。特别是随着多媒体业务的兴起，计算机已经不是单纯的处理数据的工具。这对互联网也就相应地提出了更高的要求。对那些有带宽、延迟、延迟抖动等特殊要求的应用来说，现有的“尽力而为”服务显然是不够的。

5.3.1 TCP 拥塞控制机制介绍

基于源端的拥塞控制策略中，使用最为广泛的是 TCP 协议中的拥塞控制策略，TCP 协议是目前互联网中使用最为广泛的传输协议。根据 MCI 的统计，互联网上总字节数的 95％及总数据包数的 90％使用 TCP 协议传输。 早期的 TCP 协议只有基于窗口的流控制（flow control）机制而没有拥塞控制机制，因而易导致网络拥塞。1988 年 Jacobson 针对 TCP 在网络拥塞控制方面的不足，提出了“慢启动”（Slow Start）和“拥塞避免”（Congestion Avoidance）算法。1990 年出现的 TCP Reno 版本增加了“快速重传 ”（Fast Retransmit）、“快速恢复”（Fast Recovery）算法，避免了网络拥塞不严重时采用“慢启动”算法而造成过度减小发送窗口尺寸的现象，这样 TCP 的拥塞控制就主要由这 4 个核心算法组成。近几年又出现 TCP 的改进版本如 NewReno 和选择性应答（selective acknowledgement，SACK）等。正是这些拥塞控制机制防止了今天网络的拥塞崩溃。

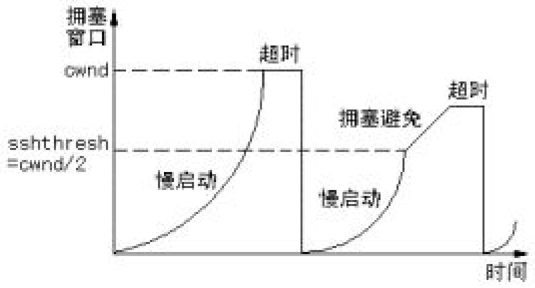
TCP 拥塞控制四个主要过程（如图 4（a）和（b）所示）简要介绍如下：

1. 慢启动阶段：早期开发的 TCP 应用在启动一个连接时会向网络中发送大量的数据包，这样很容易导致路由器缓存空间耗尽，网络发生拥塞，使得 TCP 连接的吞吐量急剧下降。由于 TCP 源端无法知道网络资源当前的利用状况，因此新建立的 TCP 连接不能一开始就发送大量数据，而只能逐步增加每次发送的数据量，以避免上述现象的发生。具体地说，当建立新的 TCP 连接时，拥塞窗口（congestion window，cwnd）初始化为一个数据包大小。源端按 cwnd 大小发送数据，每收到一个 ACK 确认，cwnd 就增加一个数据包发送量，这样 cwnd 就将随着回路响应时间（Round Trip Time，RTT）呈指数增长，源端向网络发送的数据量将急剧增加。事实上，慢启动一点也不慢，要达到每 RTT 发送 W 个数据包所需时间仅为 RTT×logW。由于在发生拥塞时，拥塞窗口会减半或降到 1，因此慢启动确保了源端的发送速率最多是链路带宽的两倍。

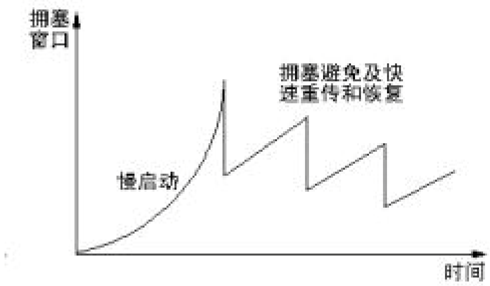
2. 拥塞避免阶段：如果 TCP 源端发现超时或收到 3 个相同 ACK 副本时，即认为网络发生了拥塞（主要因为由传输引起的数据包损坏和丢失的概率很小（<<1%））。此时就进入拥塞避免阶段。慢启动阈值（ssthresh）被设置为当前拥塞窗口大小的一半；如果超时，拥塞窗口被置 1。如果 cwnd>ssthresh，TCP 就执行拥塞避免算法，此时，cwnd 在每次收到一个 ACK 时只增加 1/cwnd 个数据包，这样，在一个 RTT 内，cwnd 将增加 1，所以在拥塞避免阶段，cwnd 不是呈指数增长，而是线性增长。

3. 快速重传和快速恢复阶段：快速重传是当 TCP 源端收到到三个相同的 ACK 副本时，即认为有数据包丢失，则源端重传丢失的数据包，而不必等待 RTO 超时。同时将 ssthresh 设置为当前 cwnd 值的一半，并且将 cwnd 减为原先的一半。快速恢复是基于“管道”模型（pipe model）的“数据包守恒”的原则（conservation of packets principle），即同一时刻在网络中传输的数据包数量是恒定的，只有当“旧”数据包离开网络后，才能发送“新”数据包进入网络。

如果发送方收到一个重复的 ACK，则认为已经有一个数据包离开了网络，于是将拥塞窗口加 1。如果“数据包守恒”原则能够得到严格遵守，那么网络中将很少会发生拥塞；本质上，拥塞控制的目的就是找到违反该原则的地方并进行修正。

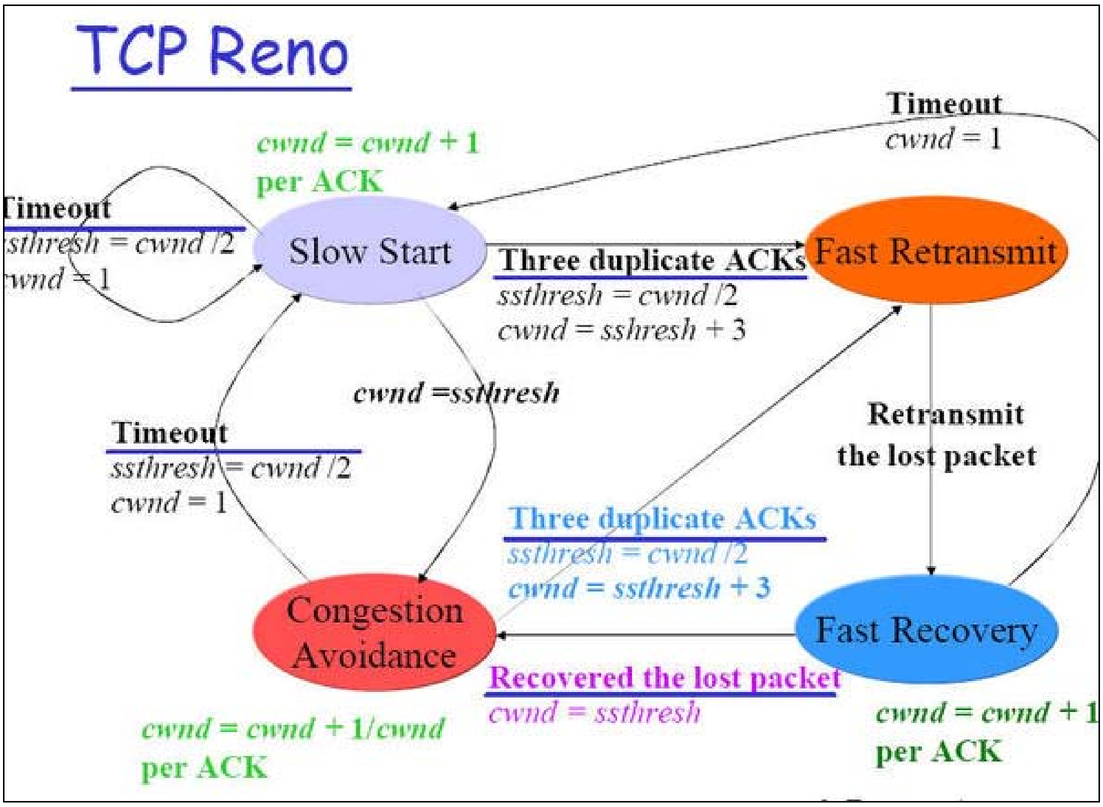


图表 5-19(a) 慢启动和拥塞避免



图表 5-20(b) 快速重传和快速恢复

经过十多年的发展，目前 TCP 协议主要包含有四个版本：TCP Tahoe、TCP Reno、TCP NewReno 和 TCP SACK。TCP Tahoe 是早期的 TCP 版本，它包括了 3 个最基本的拥塞控制算法－“慢启动”、“拥塞避免”和“快速重传”。TCP Reno 在 TCP Tahoe 基础上增加了“快速恢复”算法。TCP NewReno 对 TCP Reno 中的“快速恢复”算法进行了修正，它考虑了一个发送窗口内多个数据包丢失的情况。在 Reno 版中，发送端收到一个新的 ACK 后就退出“快速恢复”阶段，而在 NewReno 版中，只有当所有的数据包都被确认后才退出“快速恢复”阶段。TCP SACK 关注的也是一个窗口内多个数据包丢失的情况，它避免了之前版本的 TCP 重传一个窗口内所有数据包的情况，包括那些已经被接收端正确接收的数据包，而只是重传那些被丢弃的数据包。



图表 5-21 TCP Reno 的状态机

5.3.2 Linux 内核拥塞控制功能的实现

在 tcp_init 的最后调用了 tcp_register_congestion_control(&tcp_reno)，说明目前 Linux 内核缺省采用的是 reno 的拥塞控制方法。在 tcp_v4_init_sock 函数的第 33 行，我们看到有这么一句：icsk->icsk_ca_ops = &tcp_init_congestion_ops，ca 表示 congistion avoid，其中比较重要的操作函数是 tcp_reno_cong_avoid，实现如下：

```
1.
2.  /*
3.   * TCP Reno congestion control
4.   * This is special case used for fallback as well.
5.   */
6.  /* This is Jacobson's slow start and congestion avoidance.
7.   * SIGCOMM '88, p. 328.
8.   */
9.  void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 rtt, u32 in_flight,
10.          int flag)
11. {
12.     struct tcp_sock *tp = tcp_sk(sk);
13.
14.     if (!tcp_is_cwnd_limited(sk, in_flight))
15.         return;
16.
17.         /* In "safe" area, increase. */
18.     if (tp->snd_cwnd <= tp->snd_ssthresh)
19.             tcp_slow_start(tp);
20.         /* In dangerous area, increase slowly. */
21.         else if (sysctl_tcp_abc) {
22.             /* RFC3465: Appropriate Byte Count
23.              * increase once for each full cwnd acked
24.              */
25.             if (tp->bytes_acked >= tp->snd_cwnd*tp->mss_cache) {
```

```
26.                 tp->bytes_acked -= tp->snd_cwnd*tp->mss_cache;
27.                 if (tp->snd_cwnd < tp->snd_cwnd_clamp)
28.                     tp->snd_cwnd++;
29.             }
30.         } else {
31.             /* In theory this is tp->snd_cwnd += 1 / tp->snd_cwnd */
32.             if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
33.                 if (tp->snd_cwnd < tp->snd_cwnd_clamp)
34.                     tp->snd_cwnd++;
35.                 tp->snd_cwnd_cnt = 0;
36.             } else
37.                 tp->snd_cwnd_cnt++;
38.     }
39. }
```

代码段 5-23 tcp_reno_cong_avoid 函数

一般来说，重传发生在超时之后，但是如果发送端接受到 3 个以上的重复 ACK 的情况下，就应该意识到，数据丢了，需要重新传递。这个机制是不需要等到重传定时器溢出的，所以叫做**快速重传**，而重新传递以后，因为走的不是**慢启动**而是**拥塞避免算法**，所以这又叫做**快速恢复算法**。流程如下：

1. 当收到第 3 个重复的 ACK 时，将 ssthresh 设置为当前拥塞窗口 cwnd 的一半。重传丢失的报文段。设置 cwnd 为 ssthresh 加上 3 倍的报文段大小。

2. 每次收到另一个重复的 ACK 时， cwnd 增加 1 个报文段大小并发送 1 个分组（如果新的 cwnd 允许发送）。

3. 当下一个确认新数据的 ACK 到达时，设置 cwnd 为 ssthresh（在第 1 步中设置的值）。这个 ACK 应该是在进行重传后的一个往返时间内对步骤 1 中重传的确认。另外，这个 ACK 也应该 是对丢失的分组和收到的第 1 个重复的 ACK 之间的所有中间报文段的确认。这一步采用的是拥 塞避免，因为当分组丢失时我们将当前的速率减半。

第6章　Select 的实现机制

客户端要同时处理标准输入和 TCP socket　当输入阻塞时不能立即处理 socket　所以需要采用 I/O 复用技术，即应用于多路同步 I/O 模式。

以下几个方面需要用到 I/O 复用技术:

1. 一个客户同时处理多个描述符(如前所述)
2. 客户同时处理多个 socket
3. TCP 服务器同时处理监听端口和连接端口
4. 服务器同时处理 TCP 和 UDP
5. 服务器同时处理多个服务或多个协议

告知内核等待某一或某些事件发生　而后唤醒进程或线程超时返回。这是实现多个程序并行的一种手段。说到这里，我得提起 W.Richaed.Stevens 写的《UNIX 网络编程》书中提到多种实现程序并行的方法，比如信号驱动，多线程等，但为什么我要单独把 select 作为一章呢？从性能上说 select 不见得比多线程或线程池好，但是考虑到综合性能，select 作为操作系统必须支持的一个接口，比其它几种方式要好很多。

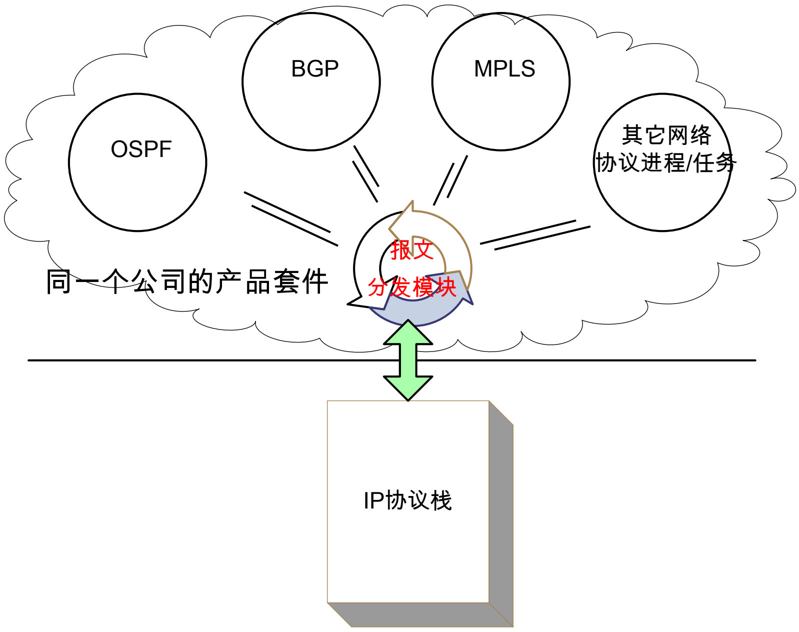
候选一：线程。虽然很多操作系统提供了基于 POSIX 的多线程接口，但是世界上还有很多嵌入式系统，没有完全支持这种接口，就算有了，那也是要多加钱去买这个模块，对于成本敏感的设备厂商，这是很痛苦的。

候选二：信号驱动。如果说线程库的移植还算比较轻松，那么信号驱动这个方法就非常难了。UNIX 和 Windows 的信号驱动接口完全不一样，很难想象要实现这样的移植要做多少工作。更难得是几百种嵌入式操作系统上实现这样的方式也不一样，对于厂商来说，这种方法几乎可以无视之。

候选三：多进程/任务。移植性还是比较困难，虽然可以做一些封装来规避这个问题，但是性能，比前面几种方式要差，因为进程/任务切换的代价比线程要大的多。

综合以上，select 就作为一种折中的方案被很多软件包提供商选为缺省的并行方式引入进来。比如 Zebra 这款比较有名的开源路由软件套件。还有一些通信业界比较著名的软件供应商，由于我不想给它们免费广告，这里就不说它们的名号了，不过我还是把这些软件包实现大致方案画在下面。其中心的“报文分发模块”是一个单独的进程/任务，它的实现基础就是 select 函数，围绕在这个中心的其它任务几乎完全受它调遣。不管是采用何种进程/任务间消息通讯方式，都达到了各模块自己运行而不用阻塞其它模块（我想大部分读者都知道 socket 收发报文是属于阻塞性质的吧）。

上文提到一个套件(suit)的概念，表示这款软件是可以根据客户的需求（或者钱）裁减功能模块的。为了达到可裁减的目的，必须把这些模块的耦合性做得尽可能低，特别是和操作系统的耦合性。select 目前几乎是最好的选择啦，大小通吃，编译通畅，价格公道，童叟无欺。实在是杀人灭口，居家旅行，必备良药啊。



图表 6-1 典型的软件包实现形式

上图是不是象一棵树？注意，在图中，所有用户层的软件协议包以及报文分发模块都是一个软件公司提供，而且基本上是一个大“模块”提供，这样可以减轻用户编译和调试的痛苦。

6.1.1 用户如何使用 select？

select 在 glic 中的函数声明如下：

int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

如果要使用 select 还必须了解以下几个宏：

FD_ZERO(*set) 清空 socket 集合

FD_SET(s, *set) 将 s 加入 socket 集合

FD_CLR(s, *set) 从 socket 集合去掉 s

FD_ISSET(s, *set) 判断 s 是否在 socket 集合中

常数 FD_SETSIZE：集合元素的最多个数

```
struct timeval{
  long tv_sec;    秒
  long tv_usec;   毫秒
};
```

对于第三个参数 timeval{}结构，一般有下面的要点：

1. 该参数用以指明等待的时间:
   a) 永远等待，则设置 NULL 指针
   b) 不等待立即返回，那么设置该结构中 2 个成员值都设为 0
   c) 等待一定时间就必须具体设置
2. 并不十分精确 最大精度 10ms 但强于 sleep()的 1s
3. 因为有 const 修饰符，所以该值不会被修改，若要计算时间 2 次调用时间函数求差时

readset writeset excepset 用以标示我们需要内核进行检测读写错误的描述符

```
1.  #include <stdio.h>
2.  #include <sys/time.h>
3.  #include <sys/types.h>
4.  #include <unistd.h>
5.
6.  int
7.  main(void)
8.  {
9.      fd_set rfds;
10.     struct timeval tv;
11.     int retval;
12.
13.     /* Watch stdin (fd 0) to see when it has input. */
14.     FD_ZERO(&rfds);
15.     FD_SET(0, &rfds);
16.
17.     /* Wait up to five seconds. */
18.     tv.tv_sec = 5;
19.     tv.tv_usec = 0;
20.     for (;;)
21.     {
        在这里 1 表示从标准输入里接收数据，比如当我运行该程序以后，随便键入普通字符，那么 select
        会接收到信号并返回 1，如果等了 5 秒以后，还没有输入，那么返回 0；如果系统内部出现问题，则返
        回－1.
22.         retval = select(1, &rfds, NULL, NULL, &tv);
23.         /* Don't rely on the value of tv now! */
24.
25.         if (retval)
26.             printf("Data is available now.\n");
27.         /* FD_ISSET(0, &rfds) will be true. */
28.
29.      }
30.      return 0;
31. }
```

代码段 6-1select 用法

select 的基本用法就是这样的，许多书已经告诉你注意第一个参数应该是 fd＋1，最后一个参数是超时设置，已经有人进行过统计，它的精度在 10 毫秒以外，即虽然 timeval 的 tv_usec 是微妙的意思，但是你填写时还是要填入一个很大的值，比如 100000，表示等待超时为 100 毫秒。应用层要注意的地方我就不多说了，现在我们来了解一下内核做了什么。

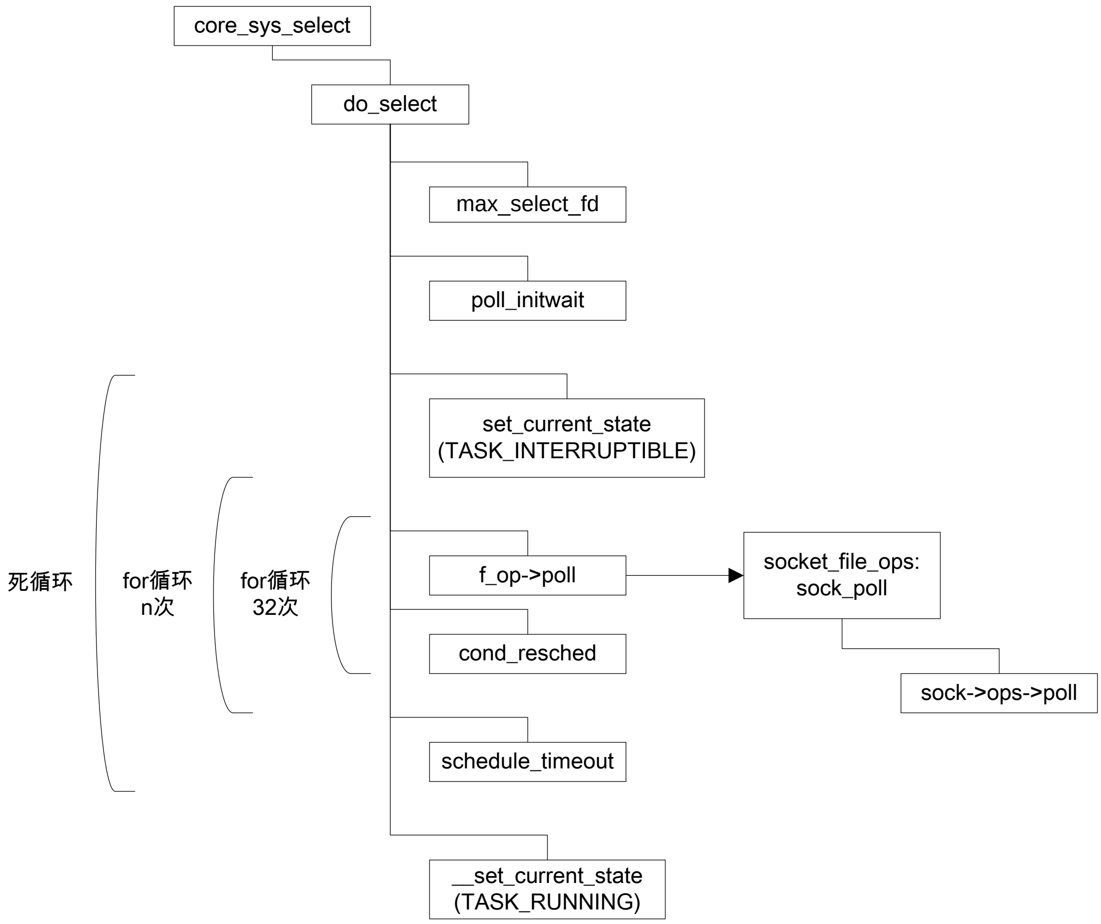
6.1.2 Select 的内核实现

很明显，select 系统调用并不只是网络使用，从系统调用的安排就可看出，它是专门有一个调用对应，而不是放在 sys_socketcall 中的一个分支。

```
1.  asmlinkage long sys_select(int n, fd_set __user *inp, fd_set __user *outp,
2.          fd_set __user *exp, struct timeval __user *tvp)
3.  {
4.      s64 timeout = -1;
5.      struct timeval tv;
6.      int ret;
7.
8.      if (tvp) {
9.          if (copy_from_user(&tv, tvp, sizeof(tv)))
10.             return -EFAULT;
11.
12.         if (tv.tv_sec < 0 || tv.tv_usec < 0)
13.             return -EINVAL;
```

```
14.
15.         /* Cast to u64 to make GCC stop complaining */
16.         if ((u64)tv.tv_sec >= (u64)MAX_INT64_SECONDS)
17.             timeout = -1;   /* infinite */
18.         else {
19.             timeout = ROUND_UP(tv.tv_usec, USEC_PER_SEC/HZ);
20.             timeout += tv.tv_sec * HZ;
21.         }
22.     }
23.
24.     ret = core_sys_select(n, inp, outp, exp, &timeout);
25.
26.     if (tvp) {
27.         struct timeval rtv;
28.
29.         if (current->personality & STICKY_TIMEOUTS)
30.             goto sticky;
31.         rtv.tv_usec = jiffies_to_usecs(do_div((*(u64*)&timeout), HZ));
32.         rtv.tv_sec = timeout;
33.         if (timeval_compare(&rtv, &tv) >= 0)
34.             rtv = tv;
35.         if (copy_to_user(tvp, &rtv, sizeof(rtv))) {
36.     sticky:
37.             /*
38.              * If an application puts its timeval in read-only
39.              * memory, we don't want the Linux-specific update to
40.              * the timeval to cause a fault after the select has
41.              * completed successfully. However, because we're not
42.              * updating the timeval, we can't restart the system
43.              * call.
44.              */
45.             if (ret == -ERESTARTNOHAND)
46.                 ret = -EINTR;
47.         }
48.     }
49.
50.     return ret;
51. }
```

代码段 6-2sys_select 函数



图表 6-2 core_sys_select 函数调用树

socket_file_ops {}结构 poll 指向 sock_poll，其会调用 sock->ops->poll，根据协议的不同，其指针分别指向：

RAW：inet_sockraw_ops→datagram_poll

UDP：inet_dgram_ops→udp_poll

TCP：inet_stream_ops→tcp_poll

其中 udp_poll 内部也会调用 datagram_poll

```
1.  /**
2.   *  datagram_poll - generic datagram poll
3.   *  @file: file struct
4.   *  @sock: socket
5.   *  @wait: poll table
6.   *
7.   *  Datagram poll: Again totally generic. This also handles
8.   *  sequenced packet sockets providing the socket receive queue
9.   *  is only ever holding data ready to receive.
10.  *
11.  *  Note: when you _don't_ use this routine for this protocol,
12.  *  and you use a different write policy from sock_writeable()
13.  *  then please supply your own write_space callback.
14.  */
15. unsigned int datagram_poll(struct file *file, struct socket *sock,
16.            poll_table *wait)
17. {
18. struct sock *sk = sock->sk;
19. unsigned int mask;
20.
```

```
21. poll_wait(file, sk->sk_sleep, wait);
22. mask = 0;
23.
24. /* exceptional events? */
25. if (sk->sk_err || !skb_queue_empty(&sk->sk_error_queue))
26.     mask |= POLLERR;
27. if (sk->sk_shutdown & RCV_SHUTDOWN)
28.     mask |= POLLRDHUP;
29. if (sk->sk_shutdown == SHUTDOWN_MASK)
30.     mask |= POLLHUP;
31.
32. /* readable? */
33. if (!skb_queue_empty(&sk->sk_receive_queue) ||
34.    (sk->sk_shutdown & RCV_SHUTDOWN))
35.     mask |= POLLIN | POLLRDNORM;
36.
37. /* Connection-based need to check for termination and startup */
38. if (connection_based(sk)) {
39.     if (sk->sk_state == TCP_CLOSE)
40.         mask |= POLLHUP;
41.     /* connection hasn't started yet? */
42.     if (sk->sk_state == TCP_SYN_SENT)
43.         return mask;
44. }
45.
46. /* writable? */
47. if (sock_writeable(sk))
48.     mask |= POLLOUT | POLLWRNORM | POLLWRBAND;
49. else
50.     set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
51.
52. return mask;
53. }
```

代码段 6-3 datagram_poll 函数

poll_wait 内部调用__pollwait，但却不是直接调用的，而是通过函数指针调用，此函数指针由内核中 poll_initwait 初始化为__pollwait，它就是将等待队列

```
1.  /* Add a new entry */
2.  static void __pollwait(struct file *filp, wait_queue_head_t *wait_address,
3.              poll_table *p)
4.  {
5.  struct poll_table_entry *entry = poll_get_entry(p);
6.
7.  entry->filp = filp;
8.  entry->wait_address = wait_address;
9.  init_waitqueue_entry(&entry->wait, current);
10. add_wait_queue(wait_address,&entry->wait);
11. }
```

代码段 6-4 __pollwait 函数

这里提一下 select 的不足：

它用 fd_set 管理所有要监视的 I/O 句柄，但是 fd_set 是一个位数组，只能接受句柄号小于 FD_SETSIZE（默认 1024）的句柄，虽然进程默认句柄号都是小于 1024 的，但是可以通过 ulimit –n 来修改，尤其是连接数超过 1024 时必需这么做（实际可能更少），如果要将大于 1024 的句柄放入 fd_set，就可能发生数组越界程序崩溃的场面。不仅如此，还有性能上的瓶颈。它们都会随着连接数的增加性能直线下降。这主要有两个原因，其一是每次 select 操作，kernel 都会建立一个当前线程关心的事件列表，并让线程阻塞在这个列表上，这是很耗时的操作。其二是每次 select 返回后，线程都要扫描所有 fd 来分发已发生的事件，这也是很耗时的。当连接数巨大时，这种消耗

积累起来，就很受不了。

下面这个内容不好单独做为一节，只好放在一个小框里提示读者吧。

> 这是 Linux 的特性，在 VxWorks5.5 中没有这样的特性。当创建一个 socket 以后，调用 setsockopt (sockfd, SOL_SOCKET, SO_BINDTODEVICE, (char *)&if_ppp0, sizeof(if_ppp0))，那么就把 sock{}→sk_bound_dev_if 指定为 eth0 的设备索引 ifindex。这样可以控制发包的接口，而且简化路由的查找。
>
> 内核里对应的函数是 sock_setsockopt，最重要的操作就是 sk->sk_bound_dev_if = dev->ifindex;。当发送报文时，会把 sk→sk_bound_dev_if 赋给相应 flowi→oif，如果查找路由表失败，那么这个 oif 会派上用场（请参考 ip_route_output_slow），我们会找到相应的设备。

第7章　2 层功能

2 层功能就是“桥”的功能。桥是比较笼统的说法，它包括了 HUB（集线器）、Switch（交换机）等一些设备，它们的共性在于使用 2 层的报文头并进行协议交互。与上两章介绍的 2 层功能不同，此 2 层不使用 ARP，那么对于服务器或 PC 领域的读者来说，这一部分是比较陌生的，但对于通信行业的读者来说，这一章就显得非常亲切了。

7.1　基本的 2 层知识

共享式网络的特点是,网络上任一瞬间只有一个节点在发送信息帧,而其他节点将接受所有的信息帧,但只是把与本节点地址相同的信息帧或广播帧拷贝下来.。

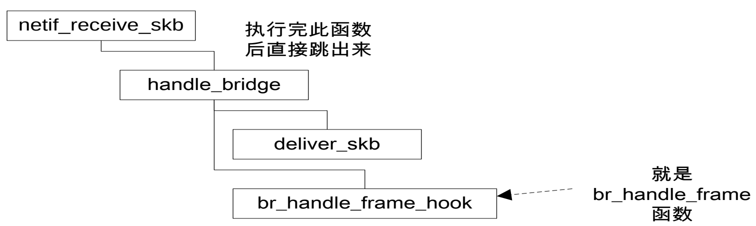
对于共享式网络,尤其共享式以太网,存在以下问题:

- 多个节点共享一段有冲突的传输介质,当网段上存在大量的用户节点时,冲突将不可能避免地发生,且冲突次数随负载的增加而增加,导致网络的性能急剧下降.虽然共享式以太网的传输速率为 10Mb/s,但实际的可用带宽只有 3.5-4.5 Mb/s,介质利用率很低.
- 随着客户端/服务器体系结构的发展,客户端需要更多地与服务器交换信息,导致网络的通信量成倍地增加,共享式网络所提供的带宽越来越不能满足不断增长的带宽需求.
- 多媒体信息,尤其是图象的适时传输,需要占用大量的带宽.共享式网络提供的带宽难于给予充分的支持.

交换式网络不是象共享式网络那样把报文分组广播到每个节点，而是在节点间沿着指定的路径传输报文分组。着相当于一个并行网络系统，多对不同源节点和不同的节点之间可以同时进行通信，而不会发生冲突，大大提高网络的可用带宽。

7.2　Linux 桥实现的基本框架

我们已经在前面分析报文的接收过程中提到了一个函数__handle_bridge，这个函数就是处理 2 层协议报文的地方，当配置了 BRIDGE 的时候，这个函数有如下功能：



图表 7-1 netif_receive_skb 调用 handle_bridge 分支

在这个函数中先判断报文所属的设备是否已经被配置为桥模式，如果没有设置，那么就返回，如果设置为桥模式了，就调用 deliver_skb 把报文传到上层，在这里上层指的是 2 层及 3 层，如果我们注册了 2 层协议处理函数，就可以调用下 2 节我们将要介绍的 2 层报文处理函数。当然，如

果是 ARP 或 IP 层报文，也会进入到此路线，当 netif_receive_skb 调用 handle_bridge 之后，就可以不用调用 deliver_skb，而直接返回了。(请自行回顾 netif_receive_skb 函数)

接下来调用 br_handle_frame_hook，它实际是一个指向 br_handle_frame 的指针，这个函数在《Linux 网络内幕中》已经介绍，我就不多说，只要记住它主要是处理 STP（生成树）协议，然后还能提供 2 层的快速转发的能力（主要指提高软件的转发能力）。

7.3 VLAN

IEEE 802.1Q 两个主要议题:

1. 桥接/交换网络
2. VLANs：VLAN: Virtual Local Area Network(虚拟局域网)，

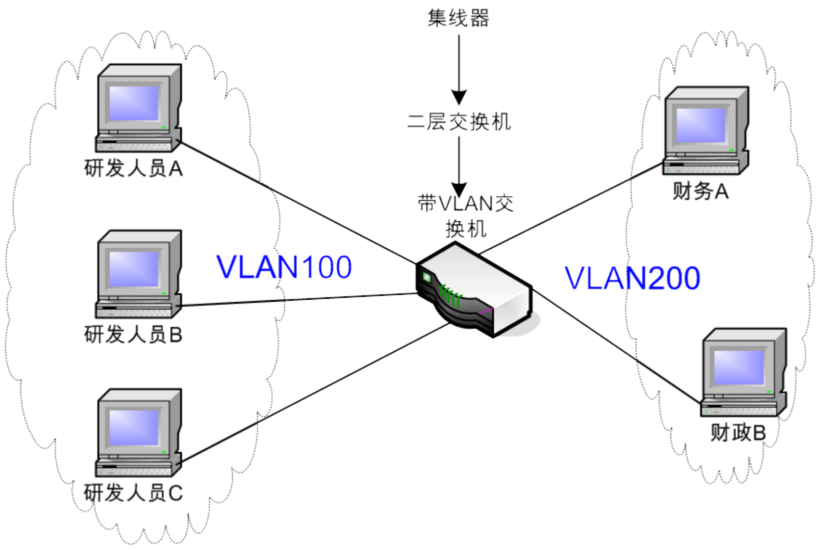
7.3.1 VLAN 概念

熟悉通信设备的读者可以跨过这一节。假设下图中的情形，某公司研发部和财务部的机器接到同一个子网里，最早期的形式是使用集线器（HUB），HUB 的特性是所有报文都广播。这会导致 2 个问题：1）安全性，研发人员会使用侦听技术把任何报文接收上来，我就曾经干过这样的事，幸运的取得了部门经理的 Email 帐号和密码，不过我只是做试验，但有的人不一定都是做试验哦。2）带宽利用率。由于广播所有报文，办公室里的网线上充斥着垃圾数据，如果形成了环路更不得了，会产生一种叫广播风暴的恶性事件。

为了解决这种事情，通信设备制造商开发出交换机（Switch），它能使报文不会随意广播，侦听技术不能派上用场了。这就是传统的二层交换机。可是如果研发部的人想跑到财务的机器上，还是非常容易，毕竟都是一个局域网中，而且，如果某个粗心的财务运行了某种狂发未知目的地址报文的软件，可能研发部的机器也不能正常工作。

为了解决这种事，人们采用 VLAN 技术来应付。理由是，如果把研发部的机器都部署到一个虚拟局域网，二财务部的机器部署到另一个虚拟局域网。这两个局域网直接不能直接通信，必须要配置路由才行。这样，财务部网络出了问题不会影响到研发部，而研发人员想进入财务部的机器可能要费点脑子了。

由于带 VLAN 功能的交换机价格便宜，而性能也不错，逐渐取代了低端路由器的市场。目前，凡是支持 VLAN 的交换机都叫做 3 层交换机，因为 VLAN 间的数据转发必须建立在 3 层路由的功能上。

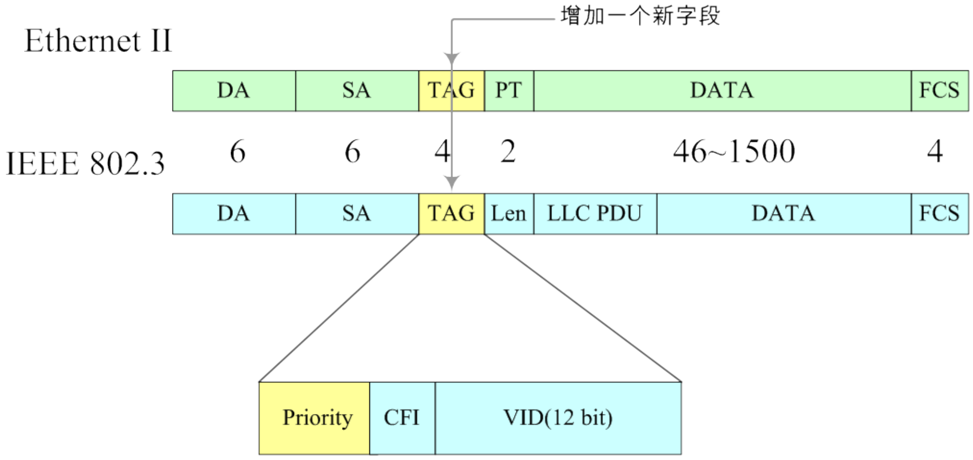


图表 7-2 VLAN 使用场景之一

（我很想把最后的那个步骤写成三层交换机，但我觉得不太严谨，因为三层交换机不一定要 VLAN 技术）

交换机的进化过程我就介绍到这里，我们来看看标准中是如何实现 VLAN 的吧。

- User_priority:优先级，从 0 到 7
- CFI:Canonical Format Indicator
- VID:VLAN 号,表示该报文属于哪一个 VLAN，由 12bit 组成，即有 4096 个 VLAN，不过 0 号 VLAN 保留，没有任何意义

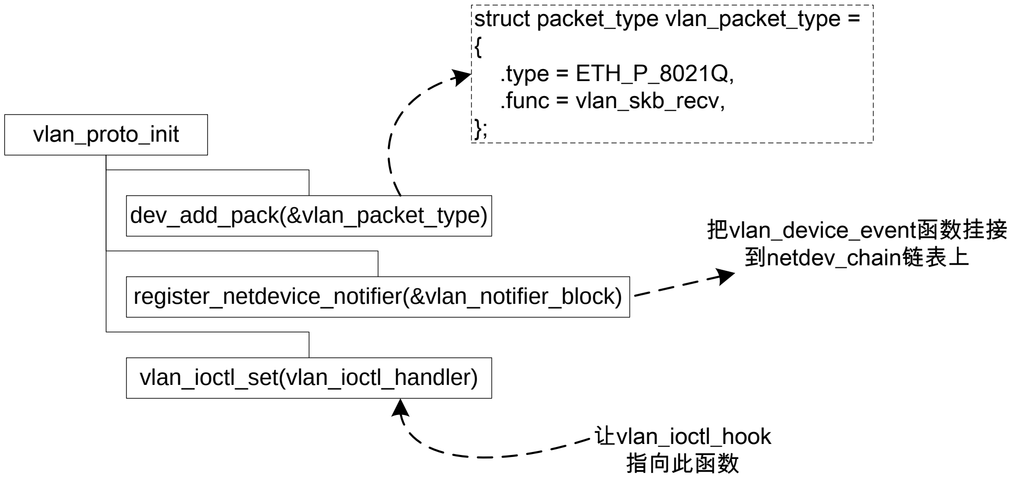


图表 7-3 VLAN 的格式

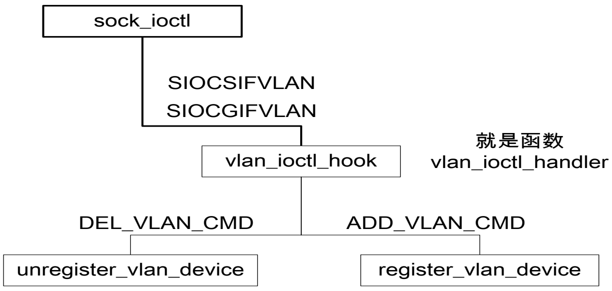
7.3.2 Linux 下 VLAN——存在巨大的缺陷

为了要使用 VLAN，在编译 Linux 内核的时候要选上该特性，因为目前 PC 机缺省情况下是不采用 VLAN 的，因为这个不仅需要 PC 的支持，还要交换机的配置来协做，比如你配置自己 PC VID 为 1，而与中间交换机的接口被管理员设置到 VID2，那么你的报文发不出去，外面的报文也不能

进入你的机器，恭喜，你不能聊天泡 MM 了。



图表 7-4 vlan_proto_init 函数调用树



图表 7-5 sock_ioctl 关于 VLAN 的分支

```
1.  /*  Attach a VLAN device to a mac address (ie Ethernet Card).
2.   *  Returns the device that was created, or NULL if there was
3.   *  an error of some kind.
4.   */
5.  static struct net_device *register_vlan_device(const char *eth_IF_name,
6.                          unsigned short VLAN_ID)
7.  {
8.  struct vlan_group *grp;
9.  struct net_device *new_dev;
10. struct net_device *real_dev; /* the ethernet device */
11. char name[IFNAMSIZ];
12.
13.
14. if (VLAN_ID >= VLAN_VID_MASK)
15.     goto out_ret_null;
16.
17. /* find the device relating to eth_IF_name. */
18. real_dev = dev_get_by_name(eth_IF_name);
19. if (!real_dev)
20.     goto out_ret_null;
21.
22. if (real_dev->features & NETIF_F_VLAN_CHALLENGED) {
23.     printk(VLAN_DBG "%s: VLANs not supported on %s.\n",
24.         __FUNCTION__, real_dev->name);
```

```
25.     goto out_put_dev;
26. }
27.
28. if ((real_dev->features & NETIF_F_HW_VLAN_RX) &&
29.     (real_dev->vlan_rx_register == NULL ||
30.      real_dev->vlan_rx_kill_vid == NULL)) {
31.     printk(VLAN_DBG "%s: Device %s has buggy VLAN hw accel.\n",
32.         __FUNCTION__, real_dev->name);
33.     goto out_put_dev;
34. }
35.
36. if ((real_dev->features & NETIF_F_HW_VLAN_FILTER) &&
37.     (real_dev->vlan_rx_add_vid == NULL ||
38.      real_dev->vlan_rx_kill_vid == NULL)) {
39.     printk(VLAN_DBG "%s: Device %s has buggy VLAN hw accel.\n",
40.         __FUNCTION__, real_dev->name);
41.     goto out_put_dev;
42. }
43.
44. /* From this point on, all the data structures must remain
45.  * consistent.
46.  */
47. rtnl_lock();
48.
49. /* The real device must be up and operating in order to
50.  * assosciate a VLAN device with it.
51.  */
52. if (!(real_dev->flags & IFF_UP))
53.     goto out_unlock;
54.
55. if (__find_vlan_dev(real_dev, VLAN_ID) != NULL) {
56.     /* was already registered. */
57.     printk(VLAN_DBG "%s: ALREADY had VLAN registered\n", __FUNCTION__);
58.     goto out_unlock;
59. }
60.
61. /* Gotta set up the fields for the device. */
62. switch (vlan_name_type) {
63. case VLAN_NAME_TYPE_RAW_PLUS_VID:
64.     /* 名字看起来象这样:   eth1.0005 */
65.     snprintf(name, IFNAMSIZ, "%s.%.4i", real_dev->name, VLAN_ID);
66.     break;
67. case VLAN_NAME_TYPE_PLUS_VID_NO_PAD:
68.     /* 把 VID 放入 VLAN 的名字中
69.      * 名字看起来象这样:   vlan5
70.      */
71.     snprintf(name, IFNAMSIZ, "vlan%i", VLAN_ID);
72.     break;
73. case VLAN_NAME_TYPE_RAW_PLUS_VID_NO_PAD:
74.      /* 名字看起来象这样:  eth0.5
75.      */
76.     snprintf(name, IFNAMSIZ, "%s.%i", real_dev->name, VLAN_ID);
77.     break;
78. case VLAN_NAME_TYPE_PLUS_VID:
79.      /* 名字看起来象这样:  vlan0005
80.      */
81. default:
82.     snprintf(name, IFNAMSIZ, "vlan%.4i", VLAN_ID);
83. };
84.
```

```
85. new_dev = alloc_netdev(sizeof(struct vlan_dev_info), name,
86.               vlan_setup);
```

对于普通以太网接口就调用 ether_setup 函数
而对于 VLAN 设备只能调用 vlan_setup

```
static void vlan_setup(struct net_device *dev)
{
    SET_MODULE_OWNER(dev);

    /* dev->ifindex = 0;  it will be set when added
to the global list.iflink is set as well.*/

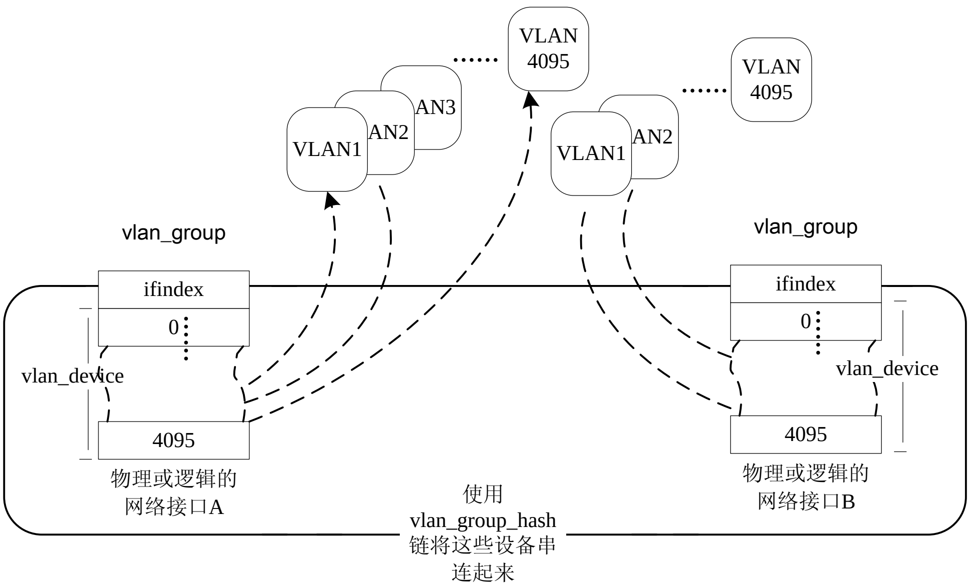
    让我们知道这是一块 VLAN“设备”
    dev->priv_flags |= IFF_802_1Q_VLAN;
    VLAN 没有自己的队列，因为底下的硬件设备能完成我们需要的一切
    dev->tx_queue_len = 0;
    dev->open = vlan_dev_open;
    dev->stop = vlan_dev_stop;
    dev->set_mac_address =
vlan_dev_set_mac_address;
    dev->destructor = free_netdev;
    dev->do_ioctl = vlan_dev_ioctl;
}
```

```
87. new_dev->flags = real_dev->flags;
88. new_dev->flags &= ~IFF_UP;
89.
90. new_dev->state = (real_dev->state & (
91.                 (1<<__LINK_STATE_DORMA
92.
93. /* need 4 bytes for extra VLAN header
94.  * hope the underlying device can han
95.  */
96. new_dev->mtu = real_dev->mtu;
97.
98. /* TODO: maybe just assign it to be E
99. new_dev->type = real_dev->type;
100.
101.        new_dev->hard_header_len = real_dev->hard_header_len;
102.        if (!(real_dev->features & NETIF_F_HW_VLAN_TX)) {
103.            /* Regular ethernet + 4 bytes (18 total). */
104.            new_dev->hard_header_len += VLAN_HLEN;
105.        }
106.        memcpy(new_dev->broadcast, real_dev->broadcast, real_dev->addr_len);
107.        memcpy(new_dev->dev_addr, real_dev->dev_addr, real_dev->addr_len);
108.        new_dev->addr_len = real_dev->addr_len;
109.
110.        if (real_dev->features & NETIF_F_HW_VLAN_TX) {
111.            new_dev->hard_header = real_dev->hard_header;
112.            new_dev->hard_start_xmit = vlan_dev_hwaccel_hard_start_xmit;
113.            new_dev->rebuild_header = real_dev->rebuild_header;
114.        } else {
115.            new_dev->hard_header = vlan_dev_hard_header;
116.            new_dev->hard_start_xmit = vlan_dev_hard_start_xmit;
117.            new_dev->rebuild_header = vlan_dev_rebuild_header;
118.        }
119.        new_dev->hard_header_parse = real_dev->hard_header_parse;
120.
121.        VLAN_DEV_INFO(new_dev)->vlan_id = VLAN_ID; /* 1 through VLAN_VID_MASK */
122.        VLAN_DEV_INFO(new_dev)->real_dev = real_dev;
123.        VLAN_DEV_INFO(new_dev)->dent = NULL;
124.        VLAN_DEV_INFO(new_dev)->flags = 1;
125.
126.        if (register_netdevice(new_dev))
127.            goto out_free_newdev;
128.
129.        lockdep_set_class(&new_dev->_xmit_lock, &vlan_netdev_xmit_lock_key);
130.
131.        new_dev->iflink = real_dev->ifindex;
132.        vlan_transfer_operstate(real_dev, new_dev);
133.        linkwatch_fire_event(new_dev); /* _MUST_ call rfc2863_policy() */
134.
135.        /* So, got the sucker initialized, now lets place
136.         * it into our local structure.
137.         */
138.        grp = __vlan_find_group(real_dev->ifindex);
139.
140.        /* Note, we are running under the RTNL semaphore
141.         * so it cannot "appear" on us.
142.         */
143.        if (!grp) { /* need to add a new group */
144.            grp = kzalloc(sizeof(struct vlan_group), GFP_KERNEL);
145.
146.            grp->real_dev_ifindex = real_dev->ifindex;
```

```
147.
148.            hlist_add_head_rcu(&grp->hlist,
149.                    &vlan_group_hash[vlan_grp_hashfn(real_dev->ifindex)]);
150.
151.            if (real_dev->features & NETIF_F_HW_VLAN_RX)
152.                real_dev->vlan_rx_register(real_dev, grp);
153.        }
154.
155.        grp->vlan_devices[VLAN_ID] = new_dev;
156.        if (real_dev->features & NETIF_F_HW_VLAN_FILTER)
157.            real_dev->vlan_rx_add_vid(real_dev, VLAN_ID);
158.
159.        rtnl_unlock();
160.
161.        return new_dev;
162.
163.    out_free_unregister:
164.        unregister_netdev(new_dev);
165.        goto out_unlock;
166.
167.    out_free_newdev:
168.        free_netdev(new_dev);
169.
170.    out_unlock:
171.        rtnl_unlock();
172.
173.    out_put_dev:
174.
175.    out_ret_null:
176.        return NULL;
177.    }
```

代码段 7-1 register_vlan_device 函数

在 vlan_group_hash 全局 hash 表有 VLAN_GRP_HASH_SIZE 个单元，也就是 2020 个单元。不知道大家从上面的片断看出 Linux 下 VLAN 有什么问题没有。



图表 7-6 VLAN“设备”组织图

从现在的实现来看有 4 个很大的缺陷：

1. 每个接口要指定给一个 VLAN 时要创建一个 net_device{}+ vlan_dev_info{}大小的设备结构给内核，如果要指定多个 VLAN，那得需要多少内存？

2. 如果系统间有 M 个接口，分别给每个接口指定 VLAN，创建 VLAN 得时候并不检查某 VLAN 是否已经创建，如果把 M 个接口指定给同一个 VLAN，则要创建 M 个相同的 VLAN，如果要指定给 N 个 VLAN，那要 M×N 个数据结构，太浪费了。

3. 从 vlan_dev_info{}结构上看，它只保存一个接口的指针，也就是说内核无法将某 VLAN 下的接口完全找到，造成了一个接口→VLAN 的单向映射：可以找到接口所属的所有 VLAN，反过来却不行。

4. 根据第 3 个问题，IP 协议栈要往某一个 VLAN 发包时，不能发到 VLAN 下的所有设备上。

让我们看看它的发送代码：

```
1.  int vlan_dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
2.  {
3.  struct net_device_stats *stats = vlan_dev_get_stats(dev);
4.  struct vlan_ethhdr *veth = (struct vlan_ethhdr *)(skb->data);
5.
6.  /* Handle non-VLAN frames if they are sent to us, for example by DHCP.
7.   *
8.   * NOTE: THIS ASSUMES DIX ETHERNET, SPECIFICALLY NOT SUPPORTING
9.   * OTHER THINGS LIKE FDDI/TokenRing/802.3 SNAPs...
10.  */
11.
12. if (veth->h_vlan_proto != __constant_htons(ETH_P_8021Q)) {
13.     int orig_headroom = skb_headroom(skb);
14.     unsigned short veth_TCI;
15.
16.     /* This is not a VLAN frame...but we can fix that! */
17.     VLAN_DEV_INFO(dev)->cnt_encap_on_xmit++;
18.
19.     /* Construct the second two bytes. This field looks something
20.      * like:
21.      * usr_priority: 3 bits (high bits)
22.      * CFI       1 bit
23.      * VLAN ID   12 bits (low bits)
24.      */
25.     veth_TCI = VLAN_DEV_INFO(dev)->vlan_id;
26.     veth_TCI |= vlan_dev_get_egress_qos_mask(dev, skb);
27.
28.     skb = __vlan_put_tag(skb, veth_TCI);
29.
30.     if (orig_headroom < VLAN_HLEN) {
31.         VLAN_DEV_INFO(dev)->cnt_inc_headroom_on_tx++;
32.     }
33. }
34.
35. stats->tx_packets++; /* for statics only */
36. stats->tx_bytes += skb->len;
37.
38. skb->dev = VLAN_DEV_INFO(dev)->real_dev;
39. dev_queue_xmit(skb);
40.
41. return 0;
42. }
```

代码段 7-2 vlan_dev_hard_start_xmit 函数

从这几份代码来看，Linux 只能支持 PC 端的 VLAN 的部分功能。不能奢望不作大修改就使 Linux 内核进入通信设备操作系统市场。

如何解决这个问题，我想从 3 个方面来解决。

第一： 要改变 vlan_dev_info{}的结构，在里面增加一个 hash 表之类的链表记录所有属于该 VLAN 的设备。这就创造了一个双向映射：接口←→VLAN。

第二： 在初始化的时候如果发现该 VLAN 结构已经创建，就不要在创建 vlan_dev_info{}了。内存消耗会减少一半。

第三： 发送报文的时候要遍历这个底层设备数据链，把报文发送给所有设备。

既然 Linux 的 VLAN 功能这么弱，我们就不再对其进行深入的研究了。

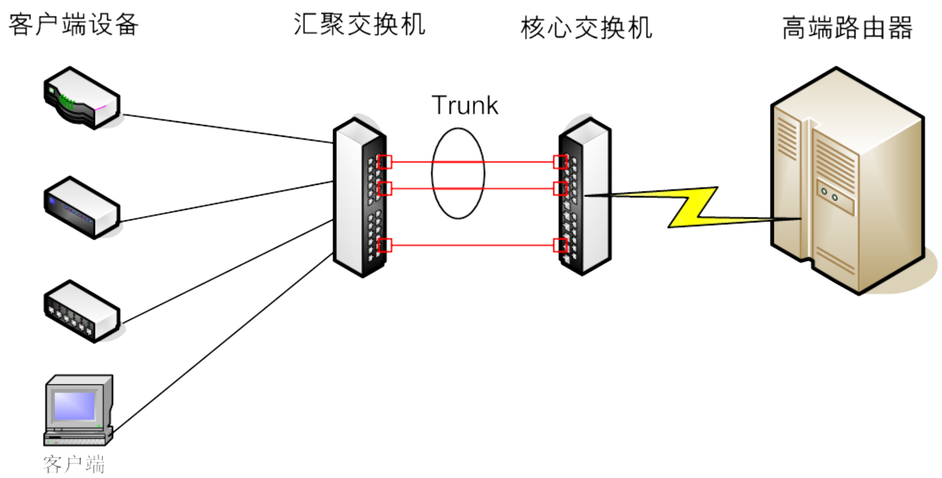
7.4 LACP 协议

LACP：Link Aggregation Control Protocol（链路聚会协议）

笔者为什么选这个协议呢？因为这个协议比较偏，不太成为热门，没有人跟我抢☺，看了《深入理解 Linux 网络内幕》，发现已经有关于 2 层协议比如 STP 的分析了。总不能再炒一遍剩饭吧。而 LACP 正好曾经是我接手参与的一个协议，当初也是参照 Linux 的内核实现进行了移植到 VxWorks 产品上，对他既然这么熟就不能不介绍了吧。

7.4.1 LACP 简介

链路聚合可以让交换机之间和交换机与服务器之间的链路带宽有非常好的伸缩性，比如可以把 2 个、3 个、4 个千兆的链路绑定在一起，使链路的带宽成倍增长。链路聚合技术可以实现不同端口的负载均衡，同时也能够互为备份，保证链路的冗余性。在这些千兆以太网交换机中，最多可以支持 4 组链路聚合，每组中最大 4 个端口。链路聚合一般是不允许跨芯片设置的。



图表 7-7 LACP 应用场景

制订于 1999 年年中的 802.3ad 标准定义了如何将两个以上的千兆以太网连接组合起来为高带宽网络连接实现负载共享、负载平衡以及提供更好的弹性。

这并不是一种新概念，许多厂商支持用于 10/100Mbps 以太网和 FDDI 的专用“干线”技术（trunking）已经有很多年了。这一标准的独特之处在于标准化实现链路聚合，802.3ad 意味着一直用于将多个低速端口组合起来形成更快的点到点逻辑链路的专用技术的终结。随着交换机和网卡厂商开始推出支持 802.3ad 的产品，你会看到在不同网卡和交换机产品中出现越来越多的兼容性，并会看到其他一些重要好处。

首先，这项标准适用于 10M、100M 和 1000Mbps 以太网。聚合在一起的链路可以在一条单一逻辑链路上组合使用上述传输速度，这就使用户在交换机之间有一个千兆端口以及 3 或 4 个 100Mbps 端口时有更多的选择，可以以负担得起的方式逐渐增加带宽。由于网络传输流被动态地

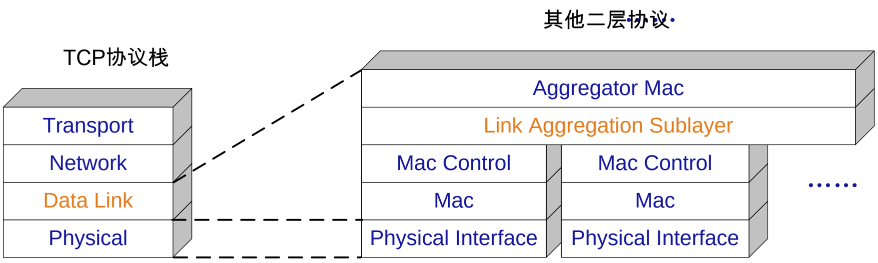
分布到各个端口，因此在聚合链路中自动地完成了对实际流经某个端口的数据的管理。

随着网络带宽需求不断地增长，可伸缩性至关重要。显然，它对于可以从添加更多千兆吞吐量中受益的骨干线路连接来说是件好事。我们不应忽视服务器的链路聚合，高性能服务器现在开始在千兆位范围内支持网络 I/O。

服务器不仅能正常地发送和接收如此巨量的数据，而且还有不少服务器具有剩余的 CPU 周期完成一些应用工作。在服务器功能变得更强、网络吞吐量需求变得更大的情况下，链路聚合为扩展留出了余地，最多可以支持 8Gbps 的全双工传输。

802.3ad 的另一个主要优点是可靠性。在链路速度可以达到 8Gbps 的情况下，链路故障将是一场灾难。关键任务交换机链路和服务器连接必须既具有强大的功能又值得信赖。即使一条电缆被误切断的情况下，它们也不会瘫痪，这正是 802.3ad 所具有的一个有趣的附带的好处。

这项链路聚合标准在点到点链路上提供了固有的、自动的冗余性。换句话说，如果链路中所使用的多个端口中的一个端口出现故障的话，网络传输流可以动态地改向链路中余下的正常状态的端口进行传输。这种改向速度很快，当交换机得知媒体访问控制地址已经被自动地从一个链路端口重新分配到同一链路中的另一个端口时，改向就被触发。然后这台交换机将数据发送到新端口位置，并且在服务几乎不中断的情况下，网络继续运行。



图表 7-8 链路聚合场景中对上层协议的影响

要聚合的端口要求

1. 端口成员类型 LAN/WAN 一致；
2. 端口成员速率，双工模式一致

加入聚合组后

3. 端口的 vlan 配置，tag 属性先恢复默认
4. 用户只能对聚合组 trunk 设置属性
5. 端口成员 vlan 配置，tag 属性要和聚合组属性一致

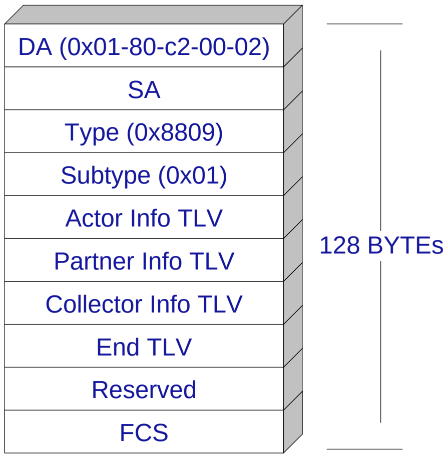
聚合方法

表格 7-1

类型	配置	LACP	特点
静态聚合	人为配置聚合组	disable	不运行状态机，仅比较本地信息，聚合速度快，可能造成业务长期丢包

动态聚合	人为配置聚合组	enable	运行状态机，比较两端信息，聚合速度慢，确保业务不丢包
自动聚合	系统自动配置聚合组	enable	无需人工配置，系统自动调节聚合

Enable lacp 协议表示设备两端运行协议状态机，利用 LACPDU 交互信息，实现聚合功能。LACP 协议通过 LACPDU 与对端交互信息。端口发送 LACPDU 向对端通告自己的系统优先级，系统 mac，端口优先级，端口号，操作 key。对端接收到这些信息后，将这些信息与本端保存相关信息进行比较以选择能够聚合的端口，从而双方可以对端口加入或退出聚合组达成一致。



图表 7-9 LACP 报文格式

LACP 是一种慢协议，据 802.3-2002 协议标准规定每秒至多发送 3 包 LACPDU。

1. 用户下发端口动态聚和配置命令
2. 端口速率，双工模式校验
3. 端口逻辑加入聚合组，各自运行状态机，发送 LACPDU 与对端端口进行信息交互
4. 当端口符合条件

   获取的对端信息和聚合组的对端信息(系统 ID,key)一致；

   本端口与聚合组本端信息(系统 ID,key)一致；

   设备两端系统 ID 不一致(确保不在同一设备聚合)；

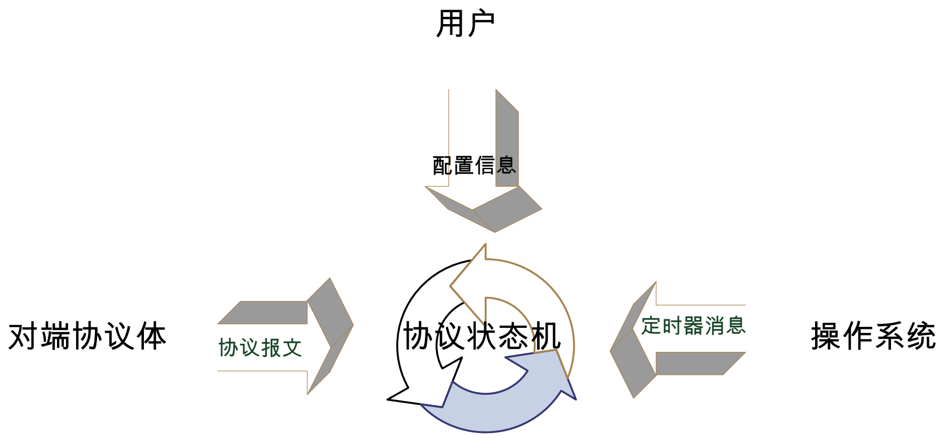
状态机运行将端口硬件物理加入聚合组，业务基于 hash 算法在物理聚合端口分发报文

LACP 协议与多种协议在数据板卡并行运行时，由于 LACP 协议会对多条物理链路进行逻辑聚合，控制物理端口的收发状态，从而对上层协议产生影响。

而 802.3 标准明确提出，LACP 协议要兼容上层协议和应用。这就涉及到 LACP 和其他协议的耦合处理问题。

7.4.2 LACP 在 Linux 中的实现

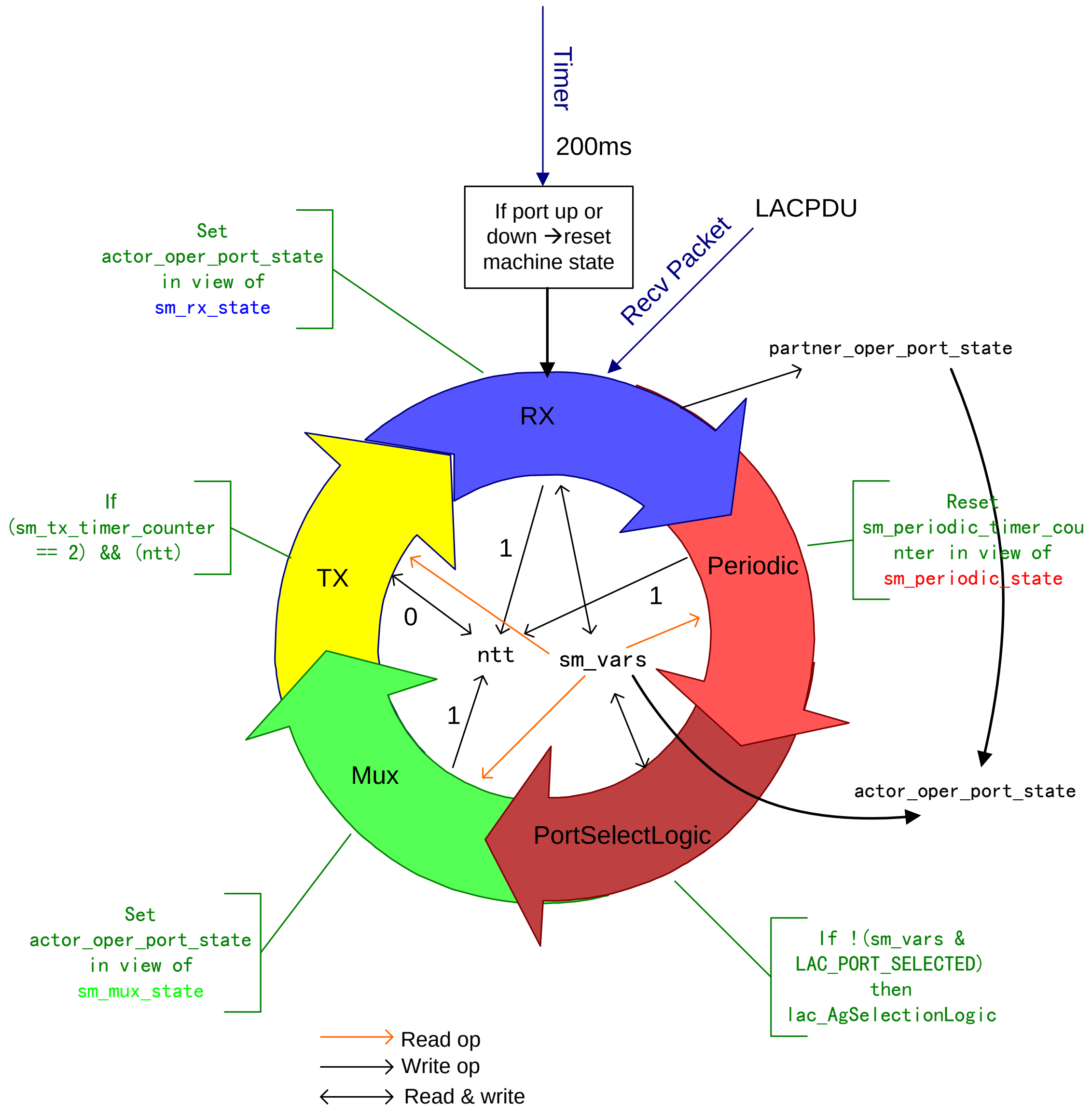
LACP 实际在通信设备产品中是以一个独立的模块实现的，而 Linux 放在内核，这会导致很多读者在研究这个协议的时候出现“头晕”的情况。由于我曾经真正把这一块拿出来仔仔细细的研究，所以对这个协议还是比较了解的。不仅如此，由于这段经历，我学习到要实现或者研究一个网络协议主要的方法。就是，不管这个协议有多复杂，它只可能有 3 个输入：



图表 7-10 协议原理图

当一个协议状态机随着系统启动而启动，它必定是受上图中 3 个激励方向中的一个或多个所推动。操作系统发出的定时器消息可以维持协议的运转，但是最终决定协议产生作用的还是用户的配置命令，大家可以回想 IP 协议栈的工作其实都始于我们对接口的配置。当对端协议体发出的协议报文通过设备及中断程序之后到达协议代码后，协议才能更进一步的运转下去。那么，我们的 LACP 也不例外。只要抓住了这三要素，解剖这个协议真是举手之劳。

下图是我在调试 Linux 内核 LACP 模块时分析得到的 LACP 状态机的运转图。要看懂这个图估计要把 802.3ag 协议和源代码结合起来，逐步地看。



图表 7-11 LACP 状态机的运转图

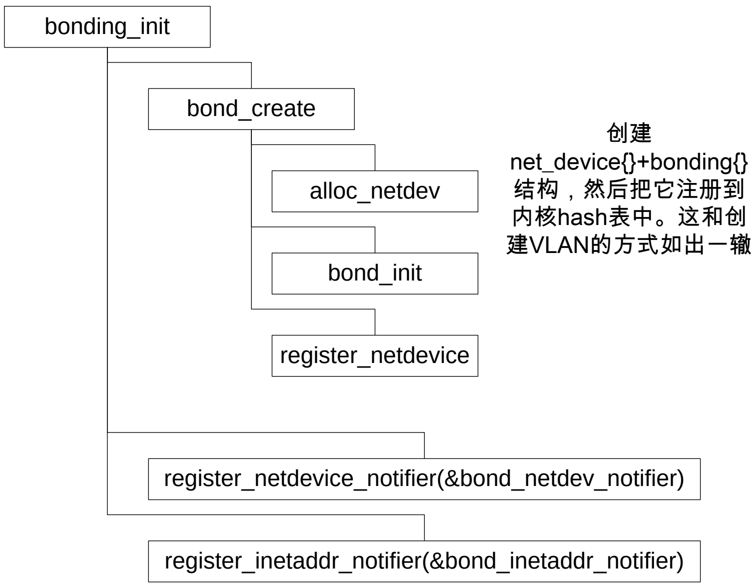
首先，要明白一个状态机要运转必须有外部激励，在大部分网络协议中外部激励主要有 3 种：

（1） 用户的配置命令

（2） 对端交流对象的数据报文

（3） 定时器

在上图中我们处于简化图示的原因，已经省略了用户配置的命令及产生的影响，但是读者一定要记住配置才是影响协议运行的最大因素。然后此图上部是 2 个外部激励：一个 200 毫秒的定时器，一个对端 LACP 实体发送过来的报文。一般我的习惯是先研究收到报文时状态机的运行方向，在此是顺时针反向。具体的处理流程大家可以参照协议标准和代码，因为那一部分是和操作系统无关，在本书中不付赘述。我展现给大家的，依然是和 Linux 联系最密切的部分。



图表 7-12 bonding_init 函数调用树

系统将 bond 的接口看成一个“设备”。在我们之前讨论实际物理设备时已经看到过 alloc_netdev 和 register_netdevice 函数，那么这个逻辑设备与物理设备不同之处在哪呢？就在下面这个 bond_init 函数中：

```
1.  static int bond_init(struct net_device *bond_dev, struct bond_params *params)
2.  {
3.  struct bonding *bond = bond_dev->priv;
4.
5.  bond->params = *params; 缺省的 bonding_defaults 参数
6.
7.  bond->first_slave = NULL;
8.  bond->curr_active_slave = NULL;
9.  bond->current_arp_slave = NULL;
10. bond->primary_slave = NULL;
11. bond->dev = bond_dev;
12. INIT_LIST_HEAD(&bond->vlan_list);
13.
14. /* Initialize the device entry points */
15. bond_dev->open = bond_open;
16. bond_dev->stop = bond_close;
17. bond_dev->get_stats = bond_get_stats;
18. bond_dev->do_ioctl = bond_do_ioctl;
19.
20. bond_dev->set_mac_address = bond_set_mac_address;
21.
22. bond_set_mode_ops(bond, bond->params.mode);
23.
24. bond_dev->destructor = free_netdev;
25.
26. /* Initialize the device options */
27. bond_dev->tx_queue_len = 0;
28. bond_dev->flags |= IFF_MASTER|IFF_MULTICAST;
29.
30. /* At first, we block adding VLANs. That's the only way to
31.  * prevent problems that occur when adding VLANs over an
32.  * empty bond. The block will be removed once non-challenged
33.  * slaves are enslaved.
34.  */
35. bond_dev->features |= NETIF_F_VLAN_CHALLENGED;
36.
37. /* don't acquire bond device's netif_tx_lock when
```

```
38.  * transmitting */
39. bond_dev->features |= NETIF_F_LLTX;
40.
41. list_add_tail(&bond->bond_list, &bond_dev_list);
42.
43. return 0;
44. }
```

代码段 7-3 bond_init 函数

上面是 bond 接口的初始化，但是它要等到用户把它 open 的时候才能正常工作。那么相应的 open 例程如下：

```
1.  static int bond_open(struct net_device *bond_dev)
2.  {
3.  struct bonding *bond = bond_dev->priv;
4.  struct timer_list *mii_timer = &bond->mii_timer;
5.  struct timer_list *arp_timer = &bond->arp_timer;
6.
7.  bond->kill_timers = 0;
8.
9.  ......
10.
11. if (bond->params.mode == BOND_MODE_8023AD) {
12.     struct timer_list *ad_timer = &(BOND_AD_INFO(bond).ad_timer);
13.     init_timer(ad_timer);
14.     ad_timer->expires  = jiffies + 1;
15.     ad_timer->data     = (unsigned long)bond;
16.     ad_timer->function = (void *)&bond_3ad_state_machine_handler;
17.     add_timer(ad_timer);
18.
        注册接收 LACPDU 报文的函数
19.     bond_register_lacpdu(bond);
20. }
21.
22. return 0;
23. }
```

```
void bond_register_lacpdu(struct bonding *bond)
{
    struct packet_type *pk_type =
&(BOND_AD_INFO(bond).ad_pkt_type);

    pk_type->type = PKT_TYPE_LACPDU;
    pk_type->dev = bond->dev;
    pk_type->func = bond_3ad_lacpdu_recv;

    dev_add_pack(pk_type);
}
```

代码段 7-4 bond_open

凡是 LACP 报文其报文头是 PKT_TYPE_LACPDU（0x8809），所以在上面注册该类报文接收函数 bond_3ad_lacpdu_recv。

上面定时器的回调函数 bond_3ad_state_machine_handler 就是 LACP 状态机主要的函数。它定时扫描物理端口的状态和选择逻辑。

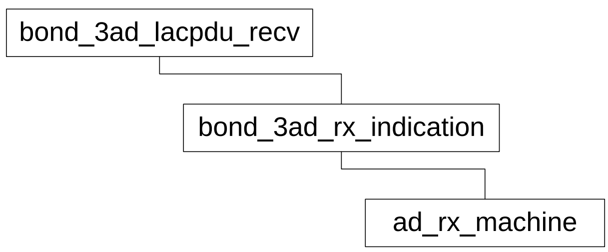
```
1.  /**
2.   * bond_3ad_state_machine_handler - handle state machines timeout
3.   * @bond: bonding struct to work on
4.   *
5.   * The state machine handling concept in this module is to check every tick
6.   * which state machine should operate any function. The execution order is
7.   * round robin, so when we have an interaction between state machines, the
8.   * reply of one to each other might be delayed until next tick.
9.   *
10.  * This function also complete the initialization when the agg_select_timer
11.  * times out, and it selects an aggregator for the ports that are yet not
12.  * related to any aggregator, and selects the active aggregator for a bond.
13.  */
14. void bond_3ad_state_machine_handler(struct bonding *bond)
15. {
16. struct port *port;
17. struct aggregator *aggregator;
18.
19. read_lock(&bond->lock);
```

```
20.
21. if (bond->kill_timers) {
22.     goto out;
23. }
24.
25. //check if there are any slaves
26. if (bond->slave_cnt == 0) {
27.     goto re_arm;
28. }
29.
30. // check if agg_select_timer timer after initialize is timed out
31. if (BOND_AD_INFO(bond).agg_select_timer
    && !(--BOND_AD_INFO(bond).agg_select_timer)) {
32.     // select the active aggregator for the bond
33.     if ((port = __get_first_port(bond))) {
34.
35.         aggregator = __get_first_agg(port);
36.         ad_agg_selection_logic(aggregator);
37.     }
38. }
39.
40. // for each port run the state machines
41. for (port = __get_first_port(bond); port; port = __get_next_port(port)) {
42.
43.     ad_rx_machine(NULL, port);
44.     ad_periodic_machine(port);
45.     ad_port_selection_logic(port);
46.     ad_mux_machine(port);
47.     ad_tx_machine(port);
48.
49.     // turn off the BEGIN bit, since we already handled it
50.     if (port->sm_vars & AD_PORT_BEGIN) {
51.         port->sm_vars &= ~AD_PORT_BEGIN;
52.     }
53. }
54.
55. re_arm:
56. mod_timer(&(BOND_AD_INFO(bond).ad_timer), jiffies + ad_delta_in_ticks);
57. out:
58. read_unlock(&bond->lock);
59. }
```

ad_delta_in_ticks ＝(AD_TIMER_INTERVAL * HZ) / 1000 在大多数系统中是 100ms。

代码段 7-5 bond_3ad_state_machine_handler 函数

在这个定时器函数中我们要注意的是 ad_rx_machine 函数，它在此传入的参数是 NULL，即没有传入报文进入到这个函数中。不仅定时器函数调用了这个函数，真正的报文接收函数 bond_3ad_lacpdu_recv 函数最终也调用了这个函数，不过此时传入的参数是对端发送的 LACP 报文。



图表 7-13 bond_3ad_lacpdu_recv 函数调用树

LACP 基本上也就介绍完毕，大家可以按照这个思路去分析 STP（生成树协议）、IGMP Snoopy 协议。

后　　记

写完这本书，我只能感叹，时间真是靠挤出来的。虽然本书错漏百出（特别是后半截），但是我还是要现在挂出来，因为我憋了三年，我的仔仔就要出来了，如果总是一拖再拖，估计我仔仔可以帮我来完成了。所以，请大家见谅，让这本书和我仔仔一起成长吧。

尽管《深入理解 Linux 网络内幕》已经面市，但我还是觉得我要继续完成这部分工作，一来了却自己的愿望，二来希望用自己的语言和见解来分析 Linux 的协议栈，使之更符合中国软件工程师的需要。我的目的就达到了。本书的名字叫《Linux 协议栈源码分析》。即看此书能大概了解 Linux 是如何实现 TCP/IP 协议栈的，但却不是能用来当枕头的"红宝书"，所以，你必须结合协议的 RFC 文档来看本书。

本人将选择 Linux 和 FreeBSD 的网络协议栈作为自己的业余研究对象。所以本套书应该有 2 本，该本是 Linux 协议栈的刨析。另一本还在计划之中。为何选择这两者？其一：代码是公开且免费的，不会有公司来找我麻烦，我也不需采用极端的方式获取这些代码。其二：这两种代码已成为网络应用之事实标准，Linux 在当前大行其道，自不必说，而 FreeBSD 是 BSD 家族中应用最广泛者，其协议栈为 VxWorks 等工业级操作系统采用，而且是故去的 steven 所著的《TCP-IP 详解_卷 2_实现》的代码基。为投开发人员之所好，我决定研究这 2 个操作系统的网络协议栈。本书就是我的第一本关于 Linux 方面的文档。将来有时间我将写出 FreeBSD 的协议栈文档。

我们知道如果想研究 BSD 的协议栈可以看 Gray R.Wright 和 W.Richard Stevens 编写的《TCP/IP Illustrated Volume2: The Implementation》（中文译名：《TCP/IP 详解 卷 2：实现》），但是 FreeBSD 已经进化到 7.0 版本了，其内部的结构已经有相当大的变化，光看那本书估计还不够。而且，根据我的经验，作为高端嵌入式软件市场占有量最大的嵌入式操作系统 VxWorks，其网络协议栈即从 BSD 发展而来，却对 BSD 协议栈进行了大量的修改，连代码风格都不一样。但看这本书就更不能完全理解 VxWorks 的网络协议栈。于是许多研究者在研究路由器操作系统的协议栈时，不知从何处研究起。

Linux 和 FreeBSD 的协议栈有很大不同，研究两者才会有对比。FreeBSD 已发展至 7.0（2008/9），其协议栈已经做了相当多的改动。我觉得应该会有很多内容可以写的。

在本文完成之际，作者谨向所有给予我指导、关心、支持和帮助的老师、Leader、同事、同学和亲人致以衷心的感谢！

参考文献

[1] W.Richard Stevens，UNIX 网络编程 卷 1 连网的 API：套接字与 XTI，清华大学出版社，1998.7.

[2] Christian Benvenuti，Understanding Linux Network Internals，2006.5

[3] Gray R.Wright 和 W.Richard Stevens TCP/IP Illustrated Volume2: The Implementation